REPORT DOCUMENTATION PAGE	Form Approved OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	27 Oct 97	

4. TITLE AND SUBTITLE
BEHAVIORAL RELATIONSHIPS BETWEEN SOFTWARE COMPONENTS

5. FUNDING NUMBERS

6. AUTHOR(S)
David S. Gibson

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
The Ohio State University

8. PERFORMING ORGANIZATION REPORT NUMBER
97-129D

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
THE DEPARTMENT OF THE AIR FORCE
AFIT/CIA, BLDG 125
2950 P STREET
WPAFB OH 45433

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION AVAILABILITY STATEMENT

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

14. SUBJECT TERMS	15. NUMBER OF PAGES 167
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT

BEHAVIORAL RELATIONSHIPS BETWEEN SOFTWARE COMPONENTS

DISSERTATION

Presented in Partial Fulfillment of the Requirements for

the Degree Doctor of Philosophy in the

Graduate School of The Ohio State University

By

David S. Gibson, B.S., M.S.

* * * * *

The Ohio State University

1997

Dissertation Committee:

Professor Bruce W. Weide, Adviser

Professor Mary Jean Harrold

Professor Timothy J. Long

Approved by

Adviser
Department of Computer
and Information Science

ABSTRACT

Building software systems from reusable software components has been a goal of software engineers for nearly three decades. Despite progress, the realization of this goal remains surprisingly elusive. Expensive hardware systems such as aircraft, communication networks, and factory assembly lines are designed so that various subsystems (both hardware and software) can be removed and replaced in order to change the performance and functionality of the overall system. In a similar manner, it should be possible to change the behavior of a component-based software system *in useful and predictable ways* by removing and replacing entire components.

In order to perform component-level maintenance, an engineer must understand not only the structural relationships but also the *behavioral* relationships among the component to be replaced, the system, and the replacement component. These behavioral relationships need to be clearly documented and available to engineers developing and maintaining component-based systems.

This dissertation presents a small set of precisely defined relationships that concisely express behavioral relationships between software components. These relationships may be used to provide implementers and maintainers with useful information about how components can and should be composed when integrated into component-based systems. Furthermore, these relationships encourage strict adherence to the well-established software engineering principles of modularity, information hiding, polymorphism, and extendibility.

The relationships described are language-independent and may be encoded in a variety of ways using modern programming languages. The dissertation describes how interface-only components, templates, inheritance, and other language mechanisms may be used to encode these relationships. Specific examples are provided in RESOLVE/Ada95, a component-based software engineering discipline that uses Ada as an implementation language.

To Elizabeth and Max

ACKNOWLEDGMENTS

I thank the current and former members of the United States Air Force who provided me with the opportunity and funding to pursue this degree for the past three years. To my advisor, Bruce Weide, I offer my deepest gratitude. Without his constant support, encouragement, enthusiasm, and good ideas, I would not have achieved this goal. I am also grateful to my reading committee members, Mary Jean Harrold and Tim Long, and to Bill Ogden for his many insightful suggestions.

I also thank Paolo Bucci and Rohit Goyal for their good friendship and encouragement over the past few years. I am especially grateful to my parents, John and Donna Gibson, for encouraging and supporting me in so many ways. Finally, I thank my wife, Cindy, for all of her love, tolerance, and help.

VITA

October 19, 1960	Born – Dayton, Ohio.
1983	B.S. Physics and Computer Science, Duke University
1986	M.S. Computer and Information Science, Trinity University, Texas
1983 - 1986	Software Development Section Chief, Air Force Electronic Warfare Center, Kelly AFB, Texas
1986 - 1990	Secure Computer Systems Analyst and Computer Security Special Applications Branch Chief, National Security Agency, Fort Meade, Maryland
1990 - 1992	Chief of Information Systems, Det. 4, Air Force Operational Test and Evaluation Center, Peterson AFB, Colorado
1992 - 1994	Instructor of Computer Science and Personnel Officer, Department of Computer Science, United States Air Force Academy, Colorado
1994 - present	Ph.D. Student, The Ohio State University, Columbus, Ohio

PUBLICATIONS

Research Publications

David S. Gibson and Bruce W. Weide. "Semantic Spaces for Specifications and Templates." *Proceedings of The Workshop on Foundations of Component-Based Systems*, Zurich, Switzerland, 1997.

Steven H. Edwards, David S. Gibson, Bruce W. Weide, and Sergey Zhupanov. "Software Component Relationships." *Proceedings of the 8th Annual Workshop on Software Reuse*, Columbus, Ohio, 1997.

David S. Gibson. *An Introduction to RESOLVE/Ada95.* Department of Computer and Information Science, The Ohio State University, Columbus, Ohio, April 1997. OSU-CISRC-4/97-TR23.

Timothy J. Long, Bruce W. Weide, Paolo Bucci, David S. Gibson, Murali Sitaraman, and Stephen H. Edwards. *Providing Intellectual Focus to CS1/CS2.* Department of Computer and Information Science, The Ohio State University, Columbus, Ohio, September 1997. OSU-CISRC-9/97-TR42.

David S. Gibson. *An Expert System for Advising Trinity University Computing and Information Sciences Majors on Course Selection.* Masters Thesis, Trinity University, San Antonio, Texas, December 1986.

Instructional Publications

David S. Gibson. *Instructional Materials For Computer Science 351 Computer System Organization.* Department of Computer Science, The United States Air Force Academy, Colorado, December 1993.

David S. Gibson. *Instructional Materials For Computer Science 355 Computer Architecture.* Department of Computer Science, The United States Air Force Academy, Colorado, May 1994.

FIELDS OF STUDY

Major Field: Computer And Information Science

Studies in:

Software Engineering	Prof. Bruce W. Weide
Computer Architecture	Prof. Dhabaleswar Panda
Theory	Prof. Kenneth J. Supowit

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Building software systems from reusable software components has been a goal of software engineers for nearly three decades. At the 1968 NATO Conference On Software Engineering, M. D. McIlroy proposed a software components industry [McI76]. Software components with well defined interfaces would be built and then reused in various software systems just as hardware components with standardized interfaces are used to construct physical systems. Despite definite progress, the realization of this goal remains surprisingly elusive [Tra95].

The motivation for component-based software engineering is even more compelling today than in the past. Today's software systems are extremely large and complex, requiring long and costly development efforts. Developing new systems, in large part by integrating existing software components (as opposed to building a system from scratch), clearly offers the potential to reduce system development time and expense. Furthermore, well designed component-based systems should be easier to maintain if software engineers are able to perform some maintenance tasks at the component-level rather than modifying individual lines of code.

Expensive hardware systems such as aircraft, communication networks, and factory assembly lines are designed so that various subsystems (both hardware and software) can be removed and replaced in order to change the performance and functionality of the overall system. Similarly, it should be possible to change the behavior of a component-based software system *in useful and predictable ways* by replacing some of the system's components with other components. If appropriate replacement components already exist, then the benefit of this approach is obvious. However, even if new components need to be developed, a systematic design and implementation approach that supports the ability to substitute a new component for an old one without making other changes to the system offers advantages over the ad hoc alternatives.

1.1 The Problem

The general problem this work addresses is the difficultly of designing and implementing component-based systems that support component-level maintenance. While other engineering disciplines successfully apply the component-based approach to building and maintaining physical systems, it has proven much more difficult to apply in software engineering. A primary reason for this difficulty is that distinct software components tend to be more tightly coupled with each other than most well-designed physical components. Furthermore, software components are often designed with extremely subtle dependencies that are not explicitly described. These dependencies may significantly complicate reasoning about program behavior [WH92].

1.1.1 Component Dependency Relationships

In current software development practice, most software components are designed to serve a specific purpose within the context of a specific software system. As a result, a component[1] may depend on other components used in a specific system. When isolated from the context of a specific system, a well-designed component still may need to depend on other components to achieve its purpose. When one component depends on another specific component as a result of its design, we refer to this type of relationship as a *design dependency* or say that one component is *coupled by design* to another.

Minimizing design dependencies (component coupling) has been recognized as a primary goal in software engineering since the early 1970's [SMC74]. By minimizing a component's dependencies on other components, we make a component easier to understand, easier to reason about, and easier to reuse in a variety of contexts. For these reasons, minimizing design dependencies is an important prerequisite for successful component-level maintenance.

When integrated into a software system, a component must be linked to other components in that system in order to serve its purpose. The final binding of one component's operations to another component's operations may take place statically when parts of the system are compiled or when pre-compiled modules are linked. Alternatively, one component's operations may be bound to the operations of another component dynamically, at run time. We refer to the dependencies that arise from integration of components into a specific system as *integration dependencies*.

One way to understand the difference between design dependencies and integration dependencies is to consider a library of reusable software components. All dependencies *between* individual components in the library are design dependencies dependencies which, in general, should be minimized for the reasons cited above.

[1]Henceforth, we will often use the term "component" in place of "software component" where it is clear from context that the component in question is a software component.

However, an individual component in the library may be built from many other components in the library. The dependencies between the sub-components composed together *within* a single library component are integration dependencies. In this case we can view a single, perhaps complex, component as a component-based system. Assuming a component and its sub-components are well-designed, there is no reason that the integration dependencies need to be minimized. In a well-designed system, the integration dependencies are just those dependencies necessary to construct the system.

1.1.2 Component Behavior

The reason for distinguishing between design dependencies and integration dependencies has to do with reasoning about the behavior exhibited by execution of component-based systems. For component-based maintenance of software to be viable, software engineers must be able to reason about how the behavior of a system changes when one component is substituted for another. That is, a maintainer of a component-based system needs to be able to determine whether substituting a new component for an old one will produce desired changes in system behavior without producing undesired changes. Reasoning about the behavior of a component-based system requires an understanding of the behavior of the system's components and how they are integrated together to form the system. However, if components are designed, implemented, and integrated into systems carefully, it is possible to reason in advance about some aspects of the post-integration behavior.

Software components that support reasoning about certain specified aspects of behavior, independent of the specific system into which they are integrated, are said to support *modular reasoning*, and potentially, *modular verification* of correctness [SW94, DL96, SG95, EHO94, EHMO91, WH92]. Modular verification is the process of formally justifying that the execution of a software component will exhibit certain specified properties when integrated into *any* system that guarantees certain specified properties in return. The guarantees of the system (or client components) may be viewed as the rules specifying legal compositions of components. When components support modular reasoning, component-level maintenance is much easier. In fact, it has been argued that component-level maintenance is technically an intractable problem without the ability to reason modularly about components [WHH94]. When reasoning modularly about the behavior of a component, design dependencies, and not integration dependencies, determine what other components must be examined and understood in order to understand specified aspects of post-integration component behavior.

As an example, consider an implementation of a stack designed to use an existing list component as its data representation. Stack operations such as `Push` and `Pop` can be implemented with list operations such as `Insert` and `Remove`. In this case, the

stack component has a design dependency on a list component. Now assume that the stack implementation is generic — it is parameterized by the type of items held by the stack. For the stack implementation to be integrated into a system, it must be instantiated with the type of item to be held on the stack. Assume that in a particular system, the stack component is instantiated with type `Message` defined in a component providing an abstract data type (ADT) for certain kinds of messages. In this particular system, the component formed by instantiating the stack with `Message` has an integration dependency, but no design dependency, on the message component.

Now suppose the stack implementation is not coupled to any particular list implementation. Instead, assume it is coupled to a component providing an abstract description of a list implementation including the structural and behavioral specifications of the `Insert` and `Remove` operations. When integrated into a particular system, the stack implementation must be linked to a component supplying a particular list implementation. (At integration time it might even be linked indirectly to one or more list implementations with final binding delayed until run time.) Within the context of a specific system, the component formed by instantiating the stack will have an integration dependency on the component supplying a particular list implementation. Here the integration dependency results directly from the design dependency. Note that the integration dependency is a much stronger coupling in this case since it requires commitment to one particular implementation whereas the design dependency does not.

In order to *fully* understand the behavior of an integrated stack component, we might need to understand aspects of the behavior of the components providing the stack item type and list implementation. However, to a great extent, we can understand the behavior exhibited by the stack operations without knowing anything about the type of items held by the stack. Furthermore, we can understand how and why the stack operations work correctly without knowing precisely how the list is implemented. We do, however, have to understand some aspects of the behavior of any list implementation that may be supplied. That is, we have to understand the ramifications of the design coupling to a component describing list behavior. The key point is that we can reason about many important aspects of the execution behavior produced by the stack component before it is integrated into a system.

1.1.3 Behavioral Relationships

Modern programming languages provide mechanisms such as interface-only components, generics (templates), inheritance, and run-time dispatching of operations that support various forms of abstraction. When used in a disciplined manner,

these and other language mechanisms can help reduce design dependencies and encode behavioral relationships between components. For example, inheritance is often used to encode the behavioral relationship of subtyping [LW94]. Use of inheritance, however, typically *increases* design dependencies between components, and thus should be used with great care. Few programming languages provide mechanisms that directly support specification of component behavior. Rare exceptions — primarily research languages — include Gypsy [AGBH77], Alphard [Sha81], and RESOLVE [SW94]. To support reasoning about program behavior, programming languages may be augmented with a behavioral specification language (see, for example, [SW94, DL96, Jon90, LvHKBO87]).

Software engineers maintaining component-based systems need to understand both when it is possible and when it is appropriate to substitute one component for another in a software system. The possibility of component substitution is determined in current programming languages by syntactic constraints. That is, if two components share a common structural interface, then it *might* be possible to substitute one for the other. The appropriateness of substituting one component for another depends on the behavioral properties of the two components and the desired changes in system behavior. Thus it is important for system maintainers to understand the behavioral relationships between components as well as the dependency relationships between components.

The purpose of studying the software component relationships described in this work is to concisely express design dependencies and behavioral relationships between software components. These relationships provide implementers and maintainers with useful information about how components may and should be linked together when integrated into component-based systems. Furthermore, these relationships support the goals of minimizing design dependencies and developing components about which it is possible to reason modularly. Thus software component relationships can aid maintainers in determining when one component may appropriately be substituted for another. In doing so, they provide a useful framework supporting maintenance of component-based software.

1.2 The Thesis

The work presented in this dissertation is based on three assumptions. First, we assume that complex software systems will be built, to a large extent, from existing software components. Second, we assume that maintenance of component based systems will be more cost-effective when performed at the component level rather than at the individual line-of-code level. Finally, we assume that large component-based systems may be built from components about which it is possible to reason modularly.

The first assumption is easily justified since component-based software reuse is already being applied in industry. Studies have demonstrated that the economics of software reuse make this approach to software development very compelling [Pre97, p. 747]. Justification of the second assumption is based on the observation that component replacement is generally easier than internal component modification when replacement components are available. When replacement components are not available, either existing components must be modified or new components developed from scratch. Both of these options are considerably more expensive than reusing existing components [Sel89, p. 222]. The third assumption seems plausible based on the research results cited earlier. To date, there are very few examples in the literature of non-trivial applications constructed from components designed specifically to support modular reasoning. However, commercial software packages developed by Joe Hollingsworth serve as proof-of-principle [Hol97]. We believe that as the importance of modular reasoning becomes more widely understood, other development efforts will further validate the third assumption.

Based on these assumptions, this dissertation addresses the following research issues.

- What relationships between software components do designers need to explicitly document to best support component-level maintenance of component-based systems?
- How can these component relationships be used to support component-level maintenance?
- How can these relationships be expressed in modern programming languages?

In answering these questions, this dissertation defends the following thesis:

> Component-level maintenance of software systems may be based on a small set of behavioral and dependency relationships between software components. Furthermore, these relationships can be encoded with the language mechanisms provided by modern programming languages, although not as easily as should be possible.

1.3 Related Research

The research presented in this dissertation builds upon RESOLVE-related research [Har90, HW91, Edw90, MW90, WOZ91, Hol92, SW94, Edw95, Wei97] performed by the Reusable Software Research Group at The Ohio State University. The RESOLVE language and discipline uniquely address many of the fundamental problems in component-based software engineering. In particular, the RESOLVE

approach supports formal behavioral specification of components and efficient component implementations about which it is possible to reason modularly. The property of modular verifiability, exhibited by RESOLVE components, is critical to reasoning about the behavior of component-based systems. Many of the commonly practiced object-oriented techniques, however, fail to support the property of modular verifiability [Sny86, Edw93].

The RESOLVE language primarily supports component adaptation through parametric polymorphism (generics). The ACTI model of software subsystems developed by Edwards provides a formal model of the semantics of parameterized (and non-parameterized) components. ACTI has been used in defining a formal semantics for RESOLVE [Edw95]. RESOLVE, and especially ACTI, have been influenced by the research into parameterized programming by Goguen[Gog84, Gog86]. The research we present in this dissertation adopts the basic component model of ACTI.

During the past decade, most research in component-based software has focused on object-oriented techniques. Widely used programming languages such as C++ [Str93] and the 1995 revision to Ada[2] [Int95b] provide language mechanisms supporting both parametric polymorphism and object-oriented techniques. Several well-known authors have written extensively about the construction of object-oriented, or "object-based", software components. Grady Booch's "Booch Components", implemented in Ada83 [Boo87], have served as the most widely adopted model for software components written in Ada. After his initial work in Ada, Booch re-implemented his component library in C++. In describing the design of his C++ components [Boo90, Boo94], Booch discussed the use of object-oriented language mechanisms and templates. Banner and Schonberg [BS92] examined implementing a software component library in Ada9X, a preliminary version of the 1995 Ada definition. Building upon this work and with concurrence from Booch, David Weller has begun implementing "The Ada 95 Booch Components" [Wel95]. Weller is currently implementing these components using Ada's new object-oriented features in a fashion similar to Booch's use of C++'s object-oriented features. Recent work by Magnus Kempe [Kem95] examining the use of Ada's new language mechanisms for implementing software components also appears to be heavily influenced by Booch's work.

As pointed out by Hollingsworth [Hol92], Booch's original Ada components fail to satisfy the goals of the RESOLVE approach. For example, Booch's polylithic components (e.g., list, tree, and graph) rely on reference semantics. As a result, systems using these components are not amenable to modular reasoning. Booch's C++ components and the Ada approaches described by Weller and Kempe also fail to use language mechanisms in a manner consistent with modular reasoning. Aside from the work of Falis discussed below, there does not appear to be any published research

[2]In this document, the term "Ada", without further qualification, is used to refer to the 1995 definition of the Ada programming language, previously known as Ada9X, and sometimes called Ada 95.

into how the new language mechanisms of Ada can be applied to the construction of modularly-verifiable component-based software.

Another well-known author who has written extensively about reusable software components is Bertrand Meyer [Mey88, Mey94]. Meyer is a leading advocate of object-oriented programming and is the principal developer of the Eiffel programming language [Mey88]. Meyer [Mey86] and Seidewitz [Sei94] have written about the relative strengths and weaknesses of inheritance and generics (parametric polymorphism). Both authors conclude that the two approaches may be used in a complementary manner. However, there appears to be very little research into the combined application of these two approaches, especially for practical imperative programming languages such as Ada and C++.

The Ada language-specific aspects of this work presented in Chapter 5 share goals similar to those of work on the development of the RESOLVE/C++ (RCPP) discipline [Wei97]. The RA95 approach presented in Chapter 5, however, differs from the RCPP approach in several aspects. First, in Chapter 4 we focus on understanding how a wide variety of language mechanisms may be used best in component-based software engineering. Second, the research primarily investigates the language mechanisms of Ada, which differ in many ways from those of C++. Third, the approach to embedding the RESOLVE language into Ada is fundamentally different from that used by RCPP.

The RCPP discipline relies heavily on the use of preprocessor macros that serve to make the "source" language of RCPP appear substantially different from normal C++. The benefits of this approach include making the RESOLVE and ACTI perspectives more explicit in the source language, hiding annoying C++ syntax, and improving maintainability by reducing source redundancy. The approach to RA95 presented in Chapter 5 does not require the use of a preprocessor. Following the RA95 discipline entails coding directly in Ada. One benefit of this approach is that RA95 uses language mechanisms of Ada largely as they were intended to be used. This approach should make explaining the rationale for RA95's use of various language mechanisms easier. Another benefit is that maintenance of RA95 code is maintenance of Ada code. Thus, analysis and maintenance tools available for Ada should be directly applicable to RA95 source code. Finally, a possible practical benefit of this approach is that RA95 may be more accessible to experienced Ada programmers than RCPP is to experienced C++ programmers.

The research presented in Chapter 5 also is related to an early exploration of mapping RESOLVE to the 1995 version of Ada by Ed Falis at Thompson Software [Fal95]. Falis' work centers around the use of the bridge and factory design patterns [GHJV95] to support run-time selection of component operations in Ada. Falis' work influenced work by Edwards on run-time selectable (Level 2) components now incorporated into RCPP. RA95 has borrowed some ideas from Falis' work, but takes a very different approach. The RA95 discipline presented in Chapter 5 does not use dynamic

binding mechanisms. The work of Falis assumes that dynamic binding will be used. While we discuss dynamic binding in Chapter 4, its incorporation into RA95 would add significant complexity and does not appear necessary to encode the component relationships presented in Chapter 3.

Another area of related RESOLVE research is the work by Joe Hollingsworth on the RESOLVE/Ada discipline based on the 1983 version of Ada (RA83) [Hol92]. Hollingsworth's research demonstrated that the RESOLVE approach could be applied using a programming language other than RESOLVE, namely, Ada83. Since the development of RA83, both RESOLVE and Ada have changed. The major change to RESOLVE has been the incorporation of many of the ideas of the ACTI model of subsystems [Edw95]. The ACTI model provides a formally-based framework for describing software components and the relationships between components. In 1995, a major revision to the Ada language was finalized and a new language standard was established. The revised Ada includes many new language mechanisms that are useful in describing components and component relationships as characterized by ACTI. While the RA95 discipline preserves many aspects of RA83, the focus of this research has been on the use of Ada's new language mechanisms and exploration of ACTI-inspired component relationships.

Many researchers have worked on formal reasoning about object-oriented programs. Most of this related work focuses on the formal definition of behavioral subtyping [CW85, LW90, LW94, SG95, DL96]. The Theta programming language [LDGM95, LCD+94], developed at MIT, incorporates ideas from this work. Theta provides separate mechanisms for type hierarchy, parametric polymorphism, and inheritance. The separation of type hierarchy from inheritance allows related types to have independent implementations and unrelated types to have related implementations [LCD+94, p. 1]. RESOLVE also provides this flexibility, but the inheritance-based type systems of Ada and C++ are more restrictive.

1.4 Organization

The remainder of this dissertation is organized as follows. In Chapter 2, we develop a framework and notation for describing behavioral relationships between software components and identify fundamental component relationships. In Chapter 3, we define a useful set of component relationships based on those identified in Chapter 2 and describe how they may be used to support component-based software development and maintenance. In Chapter 4, we discuss how the relationships described in Chapter 3 may be encoded using the language mechanisms of modern programming languages. In Chapter 5, we describe the RA95 discipline and show how the component relationships presented in Chapter 3 are encoded in RA95. Finally, Chapter 6 summarizes the research presented in this dissertation, describes the contributions made by this work, and proposes suggestions for further research.

CHAPTER 2

A MODEL OF BEHAVIORAL RELATIONSHIPS BETWEEN SOFTWARE COMPONENTS

In this chapter we develop a model of behavioral relationships between software components. These relationships serve as a basis from which we derive the more practical set of relationships described in Chapter 3. Section 2.1 discusses the requirements for interchangeable software components and motivates the need for the relations developed in the subsequent sections. Section 2.2 describes how components and their associated behavior are modeled. In Sections 2.3 and 2.4 we define conformance and dependency relations in terms of the behaviors described by components. Section 1.3 reviews related research and Section 2.5 summarizes the chapter.

2.1 Interchangeable Components

As briefly discussed in Chapter 1, one of the differences between most physical systems and software systems is the difficulty involved in replacing whole components and consistently achieving a desired effect. For example, consider a common table lamp composed of components such as a base, a switched socket, an electrical cord, a shade, and light bulb. We can change the behavior of the lamp simply by replacing one light bulb with another. Much more complex physical systems offer similar possibilities. For example, most desktop computers are easy to "upgrade" by adding new components or by replacing existing components in toto. In fact, many desktop computers are designed so it is possible to remove the central processing unit and replace it with a newer, more powerful version. We would like to be able to maintain component-based software similarly.

2.1.1 Component-Level Maintenance

Component-level maintenance involves changing the behavior of a software system in useful and predictable ways by removing and replacing software components rather than by modifying individual lines of executable code. Reasons for changing a system's behavior include:

- improving system performance,
- adding new functionality,
- adapting the system to new hardware or system software, and
- correcting defects in existing functionality.

The first two of these activities are called *perfective maintenance*. The last two are called *adaptive maintenance* and *corrective maintenance*, respectively. According to one widely-cited study of nearly 500 software projects, approximately 70% of maintenance costs — about half of the total life-cycle costs of typical software systems — are attributable to perfective and adaptive maintenance [LBSB80]. Clearly any approach that makes it easier for software engineers to improve system performance, extend system functionality, and adapt systems to new environments can have a significant impact in reducing software costs. Component-level maintenance can reduce the effort required for each of these maintenance tasks.

For component-level maintenance to be possible, software engineers must be able to answer the following question.

> Given system (or component) P which uses component Y, can component X be substituted for Y in P and maintain all of the properties that P required of Y?

In order to answer this question on a systematic basis, three issues must be addressed: the structural conformance of the new component, the behavioral conformance of the new component, and the mechanics of the substitution. We discuss conformance issues in this chapter. We discuss mechanisms supporting the mechanics of substitution in Chapter 4 and provide examples in Chapter 5.

Software components must be designed and implemented so that system maintainers can substitute one component for another and understand the effects of doing so on the system's behavior. A key challenge is to make it easier for a maintainer to achieve desired changes in system behavior without causing any undesired changes in behavior. For example, to improve execution time, we might want to replace one component with another that provides a more efficient implementation of some functionality. While it is important that the change improves system performance, it is just as important that use of the new component preserves the original system functionality. In practice, the question asked above is difficult to answer correctly and, in fact, is undecidable in the general case. Nevertheless, if software components are designed and implemented with the objective of substitutability in mind, this question is much easier to answer accurately. The principal reason for studying the component relationships defined in this dissertation is to enable software engineers to answer this question more easily.

2.1.2 The Role of Interface Specifications

Well-specified component interfaces make possible component-level maintenance of physical systems. Furthermore, standardization of interface specifications makes component-level maintenance of most physical systems commercially viable. An interface specification describes the requirements that a component must satisfy in order to interact with other "external" components, that is, other components in its environment. Note that an interface specification must address requirements on both sides of the interface. For example, in the case of a light bulb, the standardized interface specification must describe the width, depth, and threading (and other details) of a light bulb base. Such a specification places requirements both on conforming light bulbs and on lamp sockets designed to use conforming light bulbs. That is, both light bulb designers and light bulb socket designers must refer to the common interface specification in order for component-level maintenance of lamps to be possible.

The importance of well-defined interfaces for software components has been understood for many years. Many programming languages provide support for defining software component interfaces. However, most programming language support for interface specifications only addresses structural aspects of the interface, and not behavioral aspects. For some physical components, such as nuts and bolts, interface specifications only need to address structural issues. However, even for a component as simple as a light bulb, an interface specification may need to address more than purely structural requirements. The light bulb interface specification may need to specify minimum and maximum voltages and amperages required for proper bulb illumination and state that when an appropriate current is applied, the light bulb will illuminate. In the case of computer components, interface specifications clearly must describe much more than pin counts, shapes and sizes in order for components to interact successfully.

To determine if one software component may be substituted appropriately for another, both *structural conformance* and *behavioral conformance* must be addressed. Structural conformance is concerned with the names and signatures of component features. In many programming languages, the structural conformance of a component to an interface specification is determined partly by type checking. In statically-typed languages, structural conformance is checked either by the compiler or by the linker when a component is integrated into a system — prior to runtime. Behavioral conformance is concerned with whether a component's operations, when executed, will produce the desired (specified) effect. Checking behavioral conformance is, in general, much more difficult than checking structural conformance. The principal approaches to checking behavioral conformance are testing and verification techniques (both formal and informal). Both approaches have their strengths and weaknesses. Whichever approach is used (a combination of verification and testing is typically the best strategy), interface specifications that clearly describe behavioral requirements are essential.

2.1.3 Substitutability

Assume that two components X and Y have well-specified structural and behavioral interfaces describing the services they provide and the services they require from any system into which they are integrated. It would be useful to know whether X is substitutable for Y in *any* program (component-based system). Unfortunately, without knowing the specific behavior that some system expects of Y, we cannot determine if X can be substituted for Y except in a few degenerate cases. If X and Y are identical components (two different components with different names, but identical content) then we may safely conclude that X is substitutable for Y in all programs. However, in this case, there is clearly no reason to make the substitution since it should not change the behavior of the system in any respect. X and Y might also be nearly identical except that for some inputs an operation supplied by Y goes into an infinite loop whereas the corresponding operation in X does not. If we make the reasonable assumption that no "correct" program would enter an infinite loop, then we may conclude in this case that X may be substituted for Y in any (correct) program. Despite the fact that X and Y implement different behavior, the assumption ensures that no correct program would use Y or X in a way that could distinguish between the behaviors they implement.

An embedded real-time system might place timing and resource utilization requirements on the components it uses. In this case, simply comparing the functional behavior of two components is not sufficient for determining substitutability. For example, consider the case where X is identical to Y except that it has an additional operation that does not affect any state observable by any of the component's other operations. Here the behavior provided by X might appear to be a sub-behavior of that provided by Y. However, if the increased memory required to store the code of X's additional operation (whether or not it is used) exceeds the resource utilization limits that a program places on component Y, then X is not substitutable for Y in that system. Fortunately, most systems do not bump up against extremely strict limits on resource utilization and operation execution time.

In general, then, the only way to determine if component X is substitutable for component Y is within the context of a specific system where the requirements for Y, and thus for any component replacing Y, are clearly understood. The requirements a system or component has on another component may be expressed in terms of an interface specification. That is, if program P can use any component that provides the behavior described by interface specification S, then P should not be encoded to depend on a specific component, say for example, Y. If we encode P in such a way that it may be linked to any component implementing the behavior described by S, then component-level substitution becomes possible. If both X and Y conform to interface specification S, then X may be substituted for Y *with respect to* S in any program P.

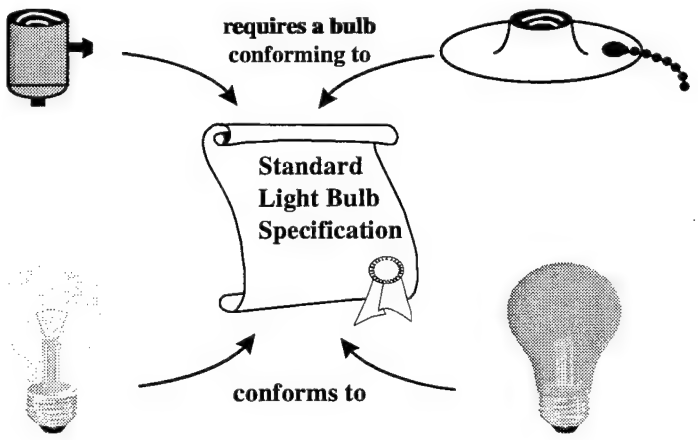


Figure 2.1: Conformance and Requirement Relationships – Physical Components

Consider, again, the analogy to a table lamp. A standard component integrated into most table lamps is a switched bulb socket. A lamp manufacturer is likely to select the kind of socket to be used in a particular lamp from a catalog of existing bulb sockets. A key factor in the selection of a socket is that it be designed to accept standard light bulbs. A description of the bulb socket selected should specifically state that the socket requires a bulb conforming to the standard light bulb specification. The socket description clearly should not require a specific brand of bulb. In addition to varying by brand, acceptable light bulbs may also vary in power consumption, radiance, color, durability, and other characteristics not fixed by the standard light bulb specification. Since many different kinds of light bulbs are designed using this specification, bulb sockets that *require* bulbs that *conform* to this specification will work with many different kinds of light bulbs.

Figure 2.1 depicts the conformance and requirement relationships involving light bulbs, light bulb sockets, and a light bulb specification. The arrows on the bottom indicate that the light bulbs shown satisfy the requirements described by the specification. The arrows on the top indicate that the bulb sockets require a light bulb (any light bulb) that conforms to the specification. Together, these two design relationships allow construction of lamps for which component-level maintenance (bulb replacement) is possible.

The analogy between physical components and software components is not perfect. Nevertheless, the role of behavioral interface specifications for software closely parallels the role of interface specifications for physical components. Figure 2.2 depicts relationships between software components analogous to those shown in Figure 2.1 for physical components. The shaded rectangular boxes at the bottom of the figure

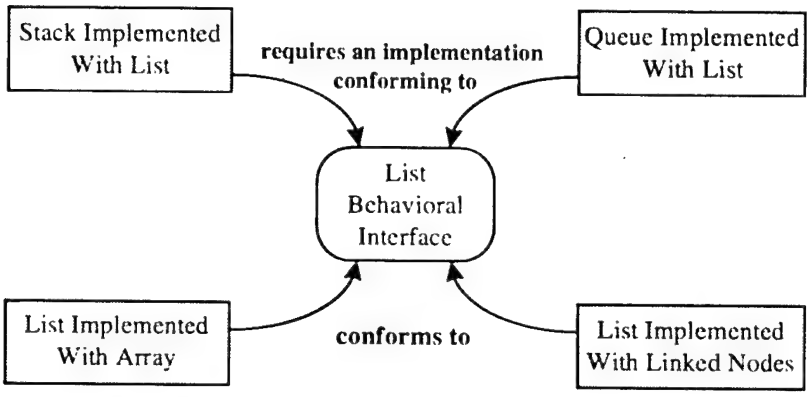


Figure 2.2: Conformance and Requirement Relationships Software Components

depict two different implementations of a list data type that *conform* to the list interface specification depicted by the oval box in the center. The two boxes at the top of the diagram depict two different components each of which *require* an implementation conforming to the list specification. The stack and queue implementations on the top are *client* components with respect to their dependency on a list implementation. Once integrated into a software system, each of these components must be linked to a specific list implementation. However, designing *and implementing* client components so that they depend on behavioral interface specifications for the components they use rather than on specific implementations, makes it possible to substitute one implementation of the specified behavior for another.

2.2 Components and Behavior

In this section we describe a model of software components and the behavior associated with the modeled components. Thus far, we have been using the term "software component" informally to refer to a unit of code which might be incorporated into an executable software system. In the model, we broaden the definition of "software component" to include behavioral interface specifications and parameterized modules called templates. The model places very few constraints on the specific form and content of components, in order to maintain language independence. But for the purposes of understanding the model, considering a component as either a specification or as an implementation of an abstract data type (ADT) will not lead the reader astray. In Chapter 4, we discuss specific ways in which components may be represented using specification and programming languages.

The model of component relationships described in this chapter consists of four disjoint sets of software components: CI (***concrete instances***), CT (***concrete templates***), AI (***abstract instances***), and AT (***abstract templates***). The model also includes the set M of ***mathematical theory modules*** (math modules, for short). For convenience, we define the set $C = CI \cup CT \cup AI \cup AT$ of all modeled components and the set $U = C \cup M$ of all modeled syntactic units. Elements of CI and CT are *concrete* components that describe how the behavior of operations is implemented. Elements of AI and AT are *abstract* components that serve as behavioral interface specifications. The components in CT and AT are *templates* while those in CI and AI are not (they are *instances*). Elements of M define mathematical theories which provide the foundation for defining the semantics of all elements of C.

All units in U must be encoded in some language L appropriate for specifying and implementing program behavior of interest. L may be a single integrated specification language such as RESOLVE [SW94] or the result of integrating independent specification and programming languages such as the approach used by Larch [GH93] and as exemplified by the RESOLVE/Ada95 components shown in Chapter 5.

The classification of software components into these four categories is based on Edwards' ACTI model of software subsystems [Edw95]. ACTI stands for ***A****bstract and* ***C****oncrete* ***T****emplates and* ***I****nstances.* However, in the ACTI model, the term "concrete instance" refers to the run-time denotation of an executable subsystem. In contrast, we use the term "concrete instance" to refer to the syntactic encoding of a component for which the run-time semantics may be modeled as an ACTI concrete instance. Similarly, we use the terms "concrete template", "abstract instance", and "abstract template" to refer to syntactic units of software whereas ACTI uses the terms to refer to a denotational semantics-based representation of the run-time behavior described by the corresponding components. The ACTI model does not define a separate unit corresponding to math modules. Instead, ACTI components include *specification adornment environments* [Edw95, p. 85] which may be used to construct components that serve the same purpose as our math modules.

The model requires that every unit in U have a unique ***unit name***. When referring to individual units, we shall use lower case identifiers such as u, m and c_1 to denote unit names. A unit's ***content*** is the string of symbols associated with a unit name and encoding a math module or a component. The only syntactic aspect of a unit's content that is modeled directly is its ***context***. The context of a unit is the finite set of all externally defined units upon which a unit directly depends.

Any unit in U may be defined directly or indirectly in terms of a finite number of *other* units in U. If one unit, say u_1 is defined in terms of another unit, say u_2, then u_1 *depends on* u_2. Components in C may depend on math modules in M and other components in C. Math modules may only be defined in terms of other math modules in M. In order for u_1 to *depend directly* on u_2, the name of unit u_2 must appear in the content of unit u_1. That is, when referring to an implementation or

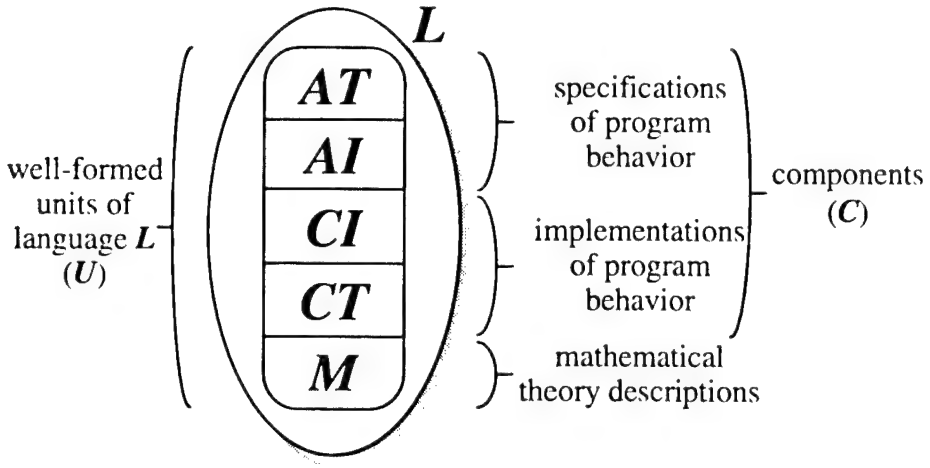


Figure 2.3: Components and Math Modules

specification element which u_1 does not itself define, u_1 must explicitly name the unit defining the referenced element. Note that no unit depends directly on itself. We refer to a unit's context by using the projection function $\mathbf{ctx}: \mathcal{U} \rightarrow \mathcal{P}_f \mathcal{U}$ (where $\mathcal{P}_f \mathcal{U}$ denotes the set of all finite subsets of $\mathcal{U}$) defined as follows:

$$\mathbf{ctx}(u) = \{u' \in \mathcal{U} - \{u\} : u \text{ depends directly on } u'\} \tag{2.1}$$

We also require that all units in $\mathcal{U}$ be *well-formed* with respect to the syntax of the language L. Only well-formed units are assigned a meaning as discussed in the following sections. Figure 2.3 depicts the five syntactic categories of units in language L discussed in this section.

2.2.1 Implementation Components

The set CI of concrete instances consists of all possible (uniquely named) implementation components that may be expressed by finite length strings in some fixed language L. A concrete instance has no parameters and represents a program unit with completely defined operations ready to integrate into a software system. For the purpose of describing elements of CI, L may be viewed as a programming language (augmented by a specification language) and element of CI as implementation modules. As noted above, every elements of CI must be *well-formed* and have a well-defined meaning. Thus, each element of CI must be a legal implementation module in accordance with the syntax of L. While a member of CI is a finite string of symbols from the finite alphabet of L, there is no upper bound on its size. Thus

any useful L supports description of a countably infinite number of concrete instances — the size of CI is $\aleph_0$. A component providing a complete data representation and fully implemented operations for manipulating a list of characters is an example of a concrete instance.

The set CT of concrete *templates* consists of all possible (uniquely named) parameterized implementation components that may be expressed as finite length strings in L. The only difference between the content of elements of CT and CI is that an element of CT refers to a single abstract instance that serves as a formal parameter for which the actual parameter is a concrete instance[3]. Whereas an element of CI models a ready-to-use component that may be incorporated directly into a larger system, an element of CT models a component that must be *instantiated* in order to *generate* a concrete instance. An implementation of a list parameterized by the type of elements contained within the list is an example of a concrete template.

The motivation for modeling concrete instances is clear — they represent the modules that make up a fully integrated component-based software system. Concrete templates also play an essential role in the model. They provide direct support for substitutability and thus are useful for more than just the generalization of families of related implementations. We discuss the primary role of concrete templates in Section 2.4.2.

2.2.2 Specification Components

In previous sections, we have referred to a behavioral interface specification as if it were different-in-kind from a software component. With physical systems, interface specifications and the physical components that conform to them do tend to be markedly different. For example, we are unlikely to confuse light bulbs and light bulb sockets with the specification document describing their standard interface. The situation is different, however, with software. Software components are symbolic *descriptions* of possible computer behavior. A software interface specification that describes *what* computer behavior is required and a software component that describes *how* some computer behavior is achieved are quite similar in nature. Both play important roles as parts (components) of a complete description of how a component-based software system may be constructed or has been constructed. To reflect the important role of software interface specifications, we include in our definition of "software component" both implementation components, members of CI and CT, *and* specification-only components, members of AI and AT.

The set AI of abstract instances consists of all possible (uniquely named) specification components that may be expressed by finite length strings in some fixed

[3]We limit a template to a single parameter for modeling convenience. However, multiple parametric dependencies may be modeled by a single formal parameter where the actual parameter is a single concrete instance that satisfies all of the requirements expressed by what would otherwise be several parameters.

language L. An abstract instance has no parameters and represents a set of ready-to-use behavioral interface specifications. For the purposes of describing elements of AI, L may be viewed as a specification language augmented by a programming language in which the syntactic (structural) interface of an implementation module may be defined independently from its implementation. Each element of AI is a well-formed specification in accordance with the syntax of L. For any useful L, the size of AI is $\aleph_0$. An element of AI specifies a behavioral interface that describes the externally visible structure (signature) and associated operation behaviors that conforming concrete instances must provide. Thus, the specification elements of L should be sufficiently expressive to describe any behaviors of interest that may be constructed with the implementation elements of L. L might need to be very powerful in order to specify, perhaps non-deterministically, the functional and temporal aspects of behaviors exhibited by complex implementations. In general, L may need to rely on higher-order logics and a wide variety of mathematical and application domain-specific theories. Software specification languages such as $\mathcal{Z}$, VDM, the Larch Shared Language, and the specification sub-language of RESOLVE are all possible candidates for the specification notation of L. A component specifying the signatures and behavior of operations manipulating a list of characters (without describing the list implementation) is an example of an abstract instance.

The set AT of abstract templates consists of all possible (uniquely named) parameterized specification components that may be expressed as finite length strings in L. The relationship between elements of AI and elements of AT is analogous to the relationship between elements of CI and CT. The only difference between the content of elements of AT and AI is that an element of AT refers to one abstract instance that serves as a formal parameter for which the actual parameter is a concrete instance. An element of AT models a generic behavioral interface that must be instantiated in order to generate an abstract instance. A specification of the signatures and behavior of operations manipulating a list that is parameterized by the type of elements contained in the list is an example of an abstract template.

As a specification independent from particular implementations, an abstract instance may serve two closely related but distinct roles. First, an abstract instance may be used to describe the behavior of one or more concrete instances which conform to it. This role is depicted in the bottom half of Figure 2.2. Second, an abstract instance may be used to describe the behavioral requirements of a component at an abstract, implementation-independent level. This role is depicted in the top half of Figure 2.2. An abstract template is primarily a convenient way of generalizing a set of closely related abstract instances.

Maintaining specification components alongside implementation components allows them to be used for structural (syntactic) conformance checking at component compilation and integration time. Specification components may also be used for

behavioral (semantic) conformance checking during the process of certifying that a particular implementation conforms to a specification, when such a claim is made.

2.2.3 Math Theory Modules

The role of mathematical theory modules is to encode mathematical objects for use in modeling program behavior. We rely on set theory to provide a foundation upon which arbitrarily complex state spaces and transitions may be built to mathematically model program behavior. Enderton describes how mathematical objects such as numbers (natural, integer, rational, and real numbers), tuples, functions, and relations may be represented with sets [End77]. For example, logicians typically represent the set of natural numbers $\{0, 1, 2, \ldots\}$ by the set $\{\emptyset, \{\emptyset\}, \{\emptyset, \{\emptyset\}\}, \ldots\}$ in which the empty set represents zero and each of the other natural numbers is represented by the set of natural numbers that precede it. Using constructive definitions such as this, it is possible to define the domains of mathematical theories (such as number theory for naturals) and then prove the *axioms* of those theories using only the axioms of set theory[4].

Mathematical theories and their associated domains, operators (functions and relations over the domain), and defining axioms are encoded in math modules (elements of M) for use by components and other math modules. We adopt the terminology defined in [HLOW94] and call the domain associated with a theory a *math type* (others use the term *sort*) and functions and predicates associated with a theory *math operations*. The motivation for this terminology is to draw an analogy between math types and operations and program types and operations. The meaning encoded by a math module is derived from an *interpretation* of the math type and math operations that it defines. An interpretation function $\mathcal{I}$ maps the math type and math operations defined by each math module to representative sets. The domain of $\mathcal{I}$ is the set of math modules M. The range of $\mathcal{I}$ is a collection of sets V, large enough to model any program behaviors of interest[5].

For example, $\mathcal{I}$ might map the natural number math type defined in a math module for number theory to the set $\{\emptyset, \{\emptyset\}, \{\emptyset, \{\emptyset\}\}, \ldots\}$, the constant math operation 0 (zero) to $\emptyset$, a math operation for successor to the set of ordered pairs $\{\langle 0, 1\rangle, \langle 1, 2\rangle, \ldots\}$ (with pairs encoded as sets), and so forth. The interpretation of this theory, as conveyed by axioms stated in the math module, is that elements of the math type represent (the mathematical concept of) natural numbers. Of course (infinitely) many other sets in V could be used to represent natural numbers. In defining the role of $\mathcal{I}$ we just require that it map each mathematical object defined

[4]Zermelo-Fraenkel set theory with the Axiom of Choice (ZFC) provides widely-accepted axioms suitable for defining much of mathematics in terms of set theory [End77, p. 253].

[5]A collection V suitably large for any modeling need is the *proper class* of all sets [End77, p. 210]

by a math module to *some* set in V for which the axioms stated in the math module can be justified, ultimately in terms of the axioms of set theory.

Using the model-based specification approach, each *program type* from which program objects (variables) may be declared is modeled by a math type. The semantics of L fixes the math models for any program types built in to L. Common built-in types for programming languages include Boolean, integer, character, and floating point scalar types as well as type constructors for static data structures such as records and arrays and for dynamic pointer-based structures. The semantics of L also must define the meaning of built-in control structures such as statement sequences, conditional statements, loops, and procedure and function calls which appear in concrete instances and concrete templates. When a component (a member of C) defines a (non-built-in) program type, the program type is associated with a math type (defined in a math module) that serves as the behavioral model for the program type. The math operations defined by the axioms and theorems of a theory associated with the math type are used to describe the behavior of program operations on objects of the program type.

2.2.4 Component Behavior

Up to this point we have used phrases such as "computer behavior", "behaviors implemented by a component", and "behaviors specified by a component" without attempting to define the term "behavior". To model conformance to behavioral interface specifications, we must include some concept of computer behavior in the model. In order to make the model as language-independent as possible, however, the model cannot be too specific about the exact form of modeled behaviors.

The standard approach to modeling computer behavior is to define a collection of states and transitions between those states. A state represents the status between transitions of the physical system (a computer or computer-controlled system) that carries out operations described by software. The transitions from state to state represent the behavior of the physical system and thus the behavior described by the software. The semantics of a programming language maps well-formed syntactic elements of the language to sets of transitions in the state space. The definition of the states, which may be expressed in terms of an abstract machine rather than a specific computer, determines the extent to which different physical behaviors can be distinguished by the semantics of a programming language.

The approach we use to characterize the semantics of software components is to assume a traditional (operational or denotational) semantics for concrete instances and then define the semantics of elements of AI, CT, and AT in terms of the semantics of elements of CI. Since a concrete instance has no unresolved external dependencies, the semantics of a concrete instance that implements a single operation may be treated like the semantics of a single complete program. The semantics of a concrete instance

that implements more than one operation may be treated as potentially interacting programs that may manipulate a common state.

We use the semantic function $\mathcal{S}$ to describe the mapping of software components (all members of C) to a set-based representation in V. Since the meaning of each program type used within components is determined by its associated mathematical model, $\mathcal{S}$ is determined in large part by the mappings of $\mathcal{I}$. For example, say component $c \in CI$ refers to program type tp which is modeled by math type TM. Then within the representation of $\mathcal{S}(c)$, objects of program type t will be represented by the set given by $\mathcal{I}(TM)$. Since all elements of C must be well-formed components, $\mathcal{S}$ is a total function with domain C. However, the range of $\mathcal{S}$ is likely to be only a small portion of V which corresponds to sets that model program behavior.

We model the behavior of a concrete instance by a single element of the collection B of *abstract behaviors* which is a subset of V. The semantic function $\mathcal{S}$ maps each element of CI to an element of B. For $c \in CI$, $\mathcal{S}(c)$ represents the *meaning* (semantics) of c in the model of behavior used to define B. The set $\mathcal{S}(c)$ represents an aggregation of lower-level semantic functions yielding the meaning of each operation defined in c, which may be defined in terms of the operations in other concrete instances upon which c depends, all of which are ultimately defined in terms of $\mathcal{I}$ and the semantics of elements of L.

The nature of an element of B depends upon the type of semantics used to define L. Consider a concrete instance $c \in CI$ which implements and provides for use to other components three operations: o_1, o_2, and o_3. Then the essence of $\mathcal{S}(c) \in B$ is the set $\{\mathcal{S}_o(o_1), \mathcal{S}_o(o_2), \mathcal{S}_o(o_3)\}$ where $\mathcal{S}_o$ defines the semantics of program operations. $\mathcal{S}(c)$ might also incorporate specification information to aid in formal verification. If $\mathcal{S}$ defines the meaning of strings of L in terms of a *denotational semantics*, then $\mathcal{S}_o(o_1)$ would correspond to a partial function from states to states. The domain of the function would represent all states in which the operation could be applied meaningfully. Application of the function would model the change in state resulting from execution of o_1. If $\mathcal{S}$ defines the meanings of strings of L in terms of an *operational semantics*, then $\mathcal{S}_o(o_1)$ would correspond to a set of sequences of states. Each sequence would correspond to all intermediate states along one possible "execution path" through o_1. The space of *ACTI concrete instances* [Edw95, p. 66-77] is one way in which B might be defined using a denotational semantics approach.

To keep the model as simple as possible, we define the semantics of abstract instances *extensionally*. We define the meaning of an abstract instance as the set of meanings of all concrete instances which conform to the behavior specified by the abstract instance. Using this definition, the range of the semantic function $\mathcal{S}$ with domain restricted to AI is the power set of B, $\mathcal{P}B$ — the set of all subsets of B. A specification $a \in AI$ may be thought of as stating a behavioral requirement. Each behavior in B either does or does not satisfy that requirement. $\mathcal{S}$ maps each specification in AI to the set of *all* behaviors in B that satisfy the specified behavioral

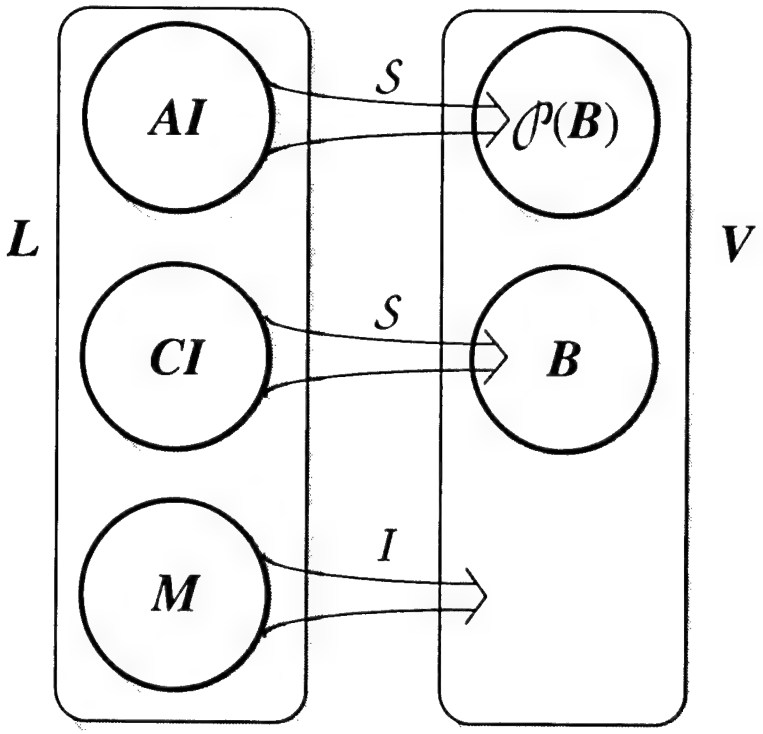


Figure 2.4: Implementations, Specifications, and Behaviors

requirement. It is possible for a specification $a \in AI$ to state a requirement that no behaviors in B satisfy. In this case, $\mathcal{S}(a) = \emptyset$. Figure 2.4 depicts all of the spaces in the model except for template components and their associated semantics. The semantics of template components will be discussed in Section 2.4.2.

Defining the semantics of a language L to the degree necessary to formally justify that an element of CI indeed conforms to an element of AI is a significant undertaking. Furthermore, once $\mathcal{S}$ and B have been defined for a specific language L, the task of verifying whether some $c \in CI$ is correct with respect to some $a \in AI$ may be extremely difficult and theoretically impossible in some cases. Nevertheless, it is possible to formally define the semantics of programming languages supporting component-based software engineering and to formalize the rules for justifying the correctness of implementations with respect to behavioral specifications [Kro88, Hey95].

2.3 Conformance Relationships

As explained in Section 2.1, in order to support component-level maintenance, we need to address the issue of an implementation's conformance to a behavioral interface specification. We would like to be able to conclude that if an implementation "conforms to" a specification, then that implementation may be used wherever there is a requirement for the behavior described by the specification. In this section we define three component conformance relations. Section 2.3.1 defines the fundamental conformance relation between concrete instances and abstract instances. Section 2.3.2 defines a conformance relation between two abstract instances.

2.3.1 Implementation-To-Specification Conformance

Informally stated, if $c \in CI$ conforms to $a \in AI$, then c must fully and correctly implement all behavior described by a. However, c may also implement behavior not specified by a as long as all requirements of a are satisfied by c. While this conformance relation is stated in terms of c and a (strings of symbols), it is the structure *and behavior* implemented by c that must conform to the structure and behavior specified by a. Using the semantic function $\mathcal{S}$ and the collection of modeled behaviors B described in Section 2.2.4, we define the conformance relation **imps**:$CI \times AI$, as follows:

$$\textbf{imps}(c, a) \equiv \mathcal{S}(c) \in \mathcal{S}(a) \tag{2.2}$$

The predicate **imps**(c, a) may be read as "component c implements component a". Figure 2.5 depicts this relationship with a solid arrow from implementation c to specification a. The behavior implemented by c, $\mathcal{S}(c)$, is represented by the point $b \in B$. The set of behaviors specified by a, $\mathcal{S}(a)$, is depicted by the single point in $\mathcal{P}B$ and by the dashed gray oval in B. The double-ended gray arrow between B and $\mathcal{P}B$ points to both representations of this set. Since, in this example, c implements a, b is a member of the set $\mathcal{S}(a)$.

imps is a many-to-many relation. Just as there are many different ways to implement a given specification, there may be many different ways to abstractly describe behavior provided by a single implementation. Implementations conforming to the same specification may vary in ways that do not affect the behavior implemented (e.g., the number of embedded comments, which might affect maintainability but not the run-time behavior). Since a conforming concrete instance may describe behavior not required by an abstract instance, implementations conforming to a common specification also may differ with respect to their implementations of these "additional" behaviors. Conforming concrete instances also may implement non-deterministically specified behaviors in ways that produce significantly different behaviors and yet still

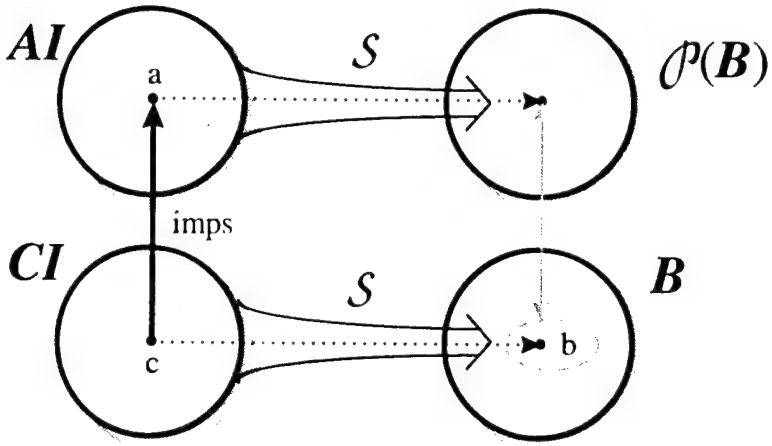


Figure 2.5: The **imps** Relation

conform to the abstract instance. The possibilities are analogous for multiple specifications that accurately characterize a single implementation. Different specifications to which a single implementation conforms may be trivially different syntactic variants that specify the same behavior. Different specifications may specify disjoint sub-behaviors of the total behavior implemented by a single conforming implementation. Finally, different specifications may describe the same implemented behavior in substantively different ways.

If **imps**(c, a) holds, then there must be some legitimate way of explaining *how* c implements a. Such an "explanation" must address how the language-specific structural (syntactic) requirements of a may be satisfied by the structure of c and how the behavioral (semantic) requirements of a may be satisfied by the operations defined by c. While providing an explanation of how c conforms to a is essential for justifying a claim that **imps**(c, a), such a claim is either valid or invalid independent of any particular explanation.

Finally, note that **imps**(c, a) does not imply that $a \in \mathbf{ctx}(c)$. That is, an implementation need not refer to a specification that it implements. From a design and implementation perspective, there are both advantages and disadvantages to having an implementation coupled to a specification to which it conforms. We will discuss these issues in Section 4.3.

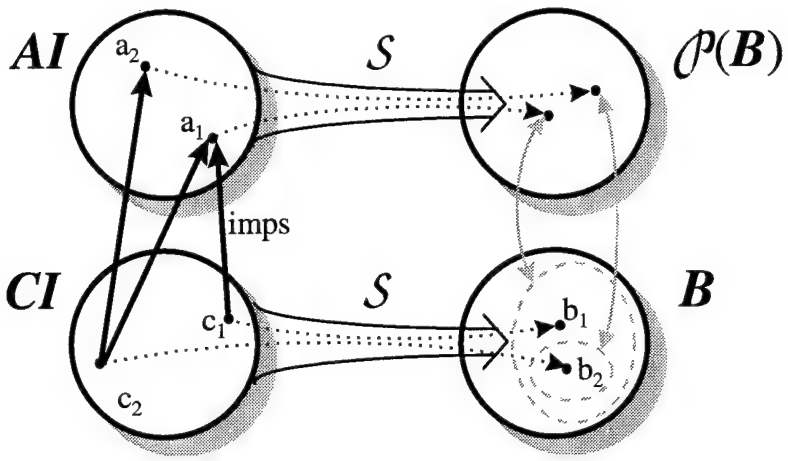


Figure 2.6: Specification Conformance And Subsets

2.3.2 Specification Extension

We now consider a conformance relationship between two specifications in AI. As discussed above, for **imps**(c, a) to hold, all behavior described by a must be implemented by c, but c may also implement additional behavior left unspecified by a. Thus c may implement a and also implement other specifications that describe more or fewer requirements on implementations than a. Consider a specification, say a_1, that places certain structural and behavioral requirements on all conforming implementations. Now assume another specification, say a_2, specifies the same behavior as a_1, plus some additional behavior not specified by a_1. In this situation any concrete instance that implements a_2 should also implement a_1, but there may be implementations of a_1 that do not implement a_2. Figure 2.6 depicts this situation.

On the left side of Figure 2.6, the solid arrows from elements of CI to elements of AI represent pairs in the **imps** relationship. That is, c_1 implements a_1 and both c_1 and c_2 implement a_2, but c_1 does not implement a_2. The property that *all* concrete instances implementing a_2 also implement a_1 (in this example) is depicted on the right side of Figure 2.6. The dashed gray ovals inside of B represent subsets of B and the double-ended gray arrows between B and $\mathcal{P}(B)$ point to two different representations of the same set of behaviors. The larger oval represents $\mathcal{S}(a_1)$, the set of all behaviors that satisfy the behavioral requirement specified by a_1. The smaller oval represents $\mathcal{S}(a_2)$, the set of all behaviors that satisfy the behavioral requirement specified by a_2. In the figure, $\mathcal{S}(a_2)$ is depicted as a subset of $\mathcal{S}(a_1)$. Therefore, *all* concrete instances that implement the behavior specified by a_2 (such as c_2) also implement the behavior

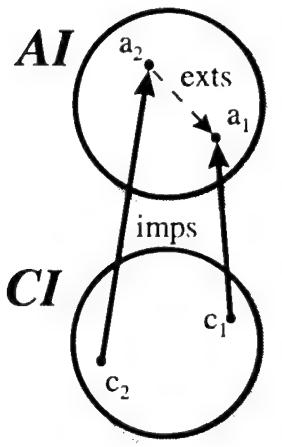


Figure 2.7: The **exts** Relation

specified by a_1. More simply stated, all concrete instances that implement a_2 also implement a_1.

In the situation described above, we may say that specification a_2 "conforms to" specification a_1. This notion of conformance is similar to that of *behavioral subtyping* [LW94]. We define this relation between two specifications, **exts**:$AI \times AI$, as follows:

$$\mathbf{exts}(a_2, a_1) \equiv \mathcal{S}(a_2) \subseteq \mathcal{S}(a_1) \tag{2.3}$$

The predicate $\mathbf{exts}(a_2, a_1)$ may be read as "specification a_2 extends specification a_1". Figure 2.7 summarizes all of the information explicitly shown in Figure 2.6. The dashed arrow from a_2 to a_1 depicts the **exts** relationship between these two specifications.

If $\mathbf{exts}(a_2, a_1)$ holds, then the behavioral requirements specified by a_2 may be viewed as an extension of the behavioral requirements specified by a_1. This does *not* mean that the (symbolic) content of a_2 is an extension of the content of a_1, although that *might* be the case. Just as c need not refer to a for $\mathbf{imps}(c, a)$ to hold, a_2 need not refer to a_1 in order for $\mathbf{exts}(a_2, a_1)$ to hold. Nevertheless, in order to justify that $\mathbf{exts}(a_2, a_1)$ holds, there must be some way of explaining *how* the behavior specified by a_2 covers all of the behavior specified by a_1.

The **exts** relation is reflexive (any specification extends itself) and transitive. It is not, however, antisymmetric (as is subset inclusion) since two *different* specification components may specify identical behavioral requirements and thus extend each other.

A useful property that follows directly from the definitions of **imps** and **exts** is:

$$\mathbf{imps}(c, a_2) \wedge \mathbf{exts}(a_2, a_1) \rightarrow \mathbf{imps}(c, a_1) \tag{2.4}$$

The **imps** relationship between c_2 and a_1 shown explicitly in Figure 2.6 is not shown in Figure 2.7, but follows immediately from the above property.

There are three basic ways in which the behavior of a specification component might be extended: *specialization*, *generalization*, and *augmentation*. If a_2 strengthens the *post-conditions* of one or more of the operations specified by a_1 and $\mathbf{exts}(a_2, a_1)$, then a_2 specializes a_1. If a_2 weakens the *pre-conditions* of one or more of the operations specified by a_1 and $\mathbf{exts}(a_2, a_1)$, then a_2 generalizes a_1. If a_2 specifies operations not specified by a_1 and $\mathbf{exts}(a_2, a_1)$, then a_2 augments a_1. Any combination of these three forms of extension (including none of them) may apply to two specifications related by the **exts** relation.

As a simple example of these three forms of extension, consider the behavioral interface specification for a bounded integer *bag* with two operations, `Insert` and `Remove`, and a maximum size of 10 integers[6]. A bag is like a set except that a bag may contain more than one occurrence of the same element. The pre-condition for `Insert` would require that the bag contain fewer than 10 integers. The post-condition of `Insert` would require that the bag contents after completion of the operation be the same as that beforehand, *except* that it should contain an additional integer of the value inserted. The pre-condition for `Remove` would require that the bag contain at least one integer. The post-condition of `Remove` would require that the contents of the bag after completion of the operation be the same as that before hand, *except* that it should contain one less integer of the value removed — some integer contained in the bag prior to execution of the `Remove` operation. This is a non-deterministic specification in that it does not specify which element of the bag is removed.

One specialization of this bag specification would be a specification that requires integers to be removed in a particular order relative to their insertion order or value. For example, a bounded integer *stack* specification might specify behavior identical to that specified by the bag except that the value of the integer removed must be the same as the value of the integer most recently inserted. Thus the post-condition on the stack operation corresponding to `Remove` would place a compatible, but stronger requirement than that of the bag on all conforming implementations. A generalization of the bag specification would be one that requires the same behavior except that the bag may hold up to 20 integers. In this case, the pre-condition for the operation corresponding to `Insert` would place a compatible, but weaker requirement on all conforming implementations. Finally an augmentation of the bag specification might specify the same requirements except that it also requires an additional operation that returns the number of elements currently in the bag. In each of these three examples,

[6]The bag specification described here was selected for simplicity and should not be interpreted as a good interface design.

any implementation that satisfies the requirements of the extended specification also satisfies the requirements of the original bag specification.

As discussed in Section 2.1.1, a common reason for changing the behavior of a software system is to add new functionality. As we discuss in Section 3.5, the augmentation form of specification extension is particularly useful for modeling extensions to component functionality.

2.4 Dependency Relationships

The three conformance relations defined in Section 2.3, **imps**, **exts**, and **exti**, are defined in terms of the semantic properties of components. These relations model useful behavioral relationships between components. The fixed dependency relation described in Section 2.4.1 is defined in terms of the syntactic properties of components and is completely orthogonal to the conformance relations. This relationship models the usual notion of component coupling applied to both implementation and specification components. The deferred dependency relation described in Section 2.4.2 models "behavioral dependencies" which directly support component-level maintenance and require the introduction of concrete templates.

2.4.1 Fixed Dependencies

In Section 2.2 we noted that a unit (a component or math module) may be defined directly or indirectly in terms of a finite number of other components. Furthermore, our definition of a "component" requires that all direct dependencies be part of a component's context. Using the **ctx** function (Equation 2.1) we now define the general coupling relation over all *components*. The relation **uses**:$C \times C$, is defined recursively as follows:

$$\begin{aligned} \mathbf{uses}(c_1, c_2) \quad \equiv \quad & c_1 = c_2 \vee \\ & c_2 \in \mathbf{cxt}(c_1) \vee \\ & \exists c \in \mathbf{cxt}(c_1) \mid \mathbf{uses}(c, c_2) \end{aligned} \tag{2.5}$$

The predicate $\mathbf{uses}(c_1, c_2)$ may be read as "component c_1 uses component c_2". In the first disjunct of the definition, c_1 and c_2 denote the same component (not two different components with the same content). In the second case, c_1 depends directly on c_2. In the recursive case (which is *not* mutually exclusive with either of the first two cases) c_1 depends directly on some component that depends either directly or indirectly on (or is) c_2.

The **uses** relation is reflexive and transitive. It is reflexive to model the possibility of features defined within a component being defined in terms of other features defined within the same component. A component may not be a member of its own context

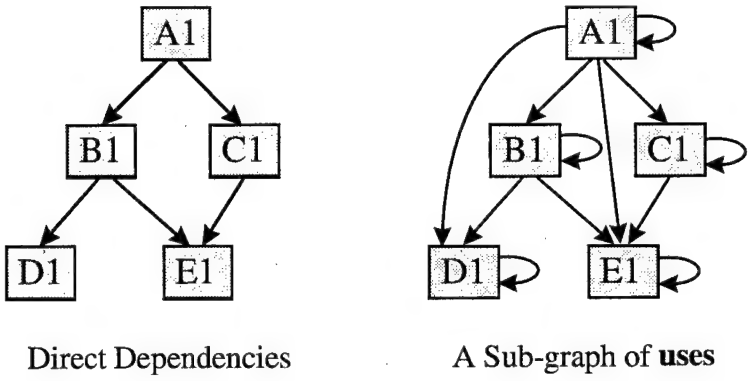


Figure 2.8: Concrete Instances Forming A Component-Based System

since **ctx** is used to express a component's external dependencies. However, if **ctx** were a reflexive relation, then **uses** simply would be the transitive closure of **ctx**. Note that two components may depend mutually upon each other (e.g., $\mathbf{uses}(c_1, c_2)$ and $\mathbf{uses}(c_2, c_1)$) may both hold) as long as components c_1 and c_2 are well-formed according to the rules of L.

If c_1 uses c_2, then component c_1 in some way depends upon component c_2. If c_1 and c_2 are both concrete instances, then operations implemented in c_1 might invoke, either directly or indirectly, operations implemented in c_2. If c_2 is a specification, then c_1 may depend on all or part of c_2 to explain behavior that it uses, extends, or implements (if c_1 is an implementation). To fully understand and justify properties about the behavior described (implemented or specified) by component c, a software engineer may need to understand the behavior described by all other components (both implementations and specifications) used by c.

Once a component-based system has been fully integrated, the behavior of the system depends only on concrete instances. We can characterize the inter-component dependencies in a fully integrated system solely in terms of the **uses** relation restricted to concrete instances. Figure 2.8 shows two views of a simple component-based system composed of five concrete instances: `A1`, `B1`, `C1`, `D1`, and `E1`, all members of CI. On the left side of Figure 2.8, the arrows between components represent direct dependencies. Thus we may conclude that that $\mathbf{ctx}(\texttt{A1}) = \{\texttt{B1}, \texttt{C1}\}$, $\mathbf{ctx}(\texttt{B1}) = \{\texttt{D1}, \texttt{E1}\}$, and $\mathbf{ctx}(\texttt{C1}) = \{\texttt{E1}\}$. Component `A1` corresponds to a "main program" and components `D1` and `E1` are components implemented entirely in terms of built-in features of L. The right side of Figure 2.8 shows the sub-graph of the **uses** relation induced by just these five concrete instances. An arrow from component c_1 to component c_2 indicates that the predicate $\mathbf{uses}(c_1, c_2)$ holds.

If **uses**(c_1, c_2) holds, then component c_2 is "hard wired" to component c_1. There is a *fixed* dependency of c_1 on c_2 that cannot be changed.[7] Any system using c_1 must also use c_2 as a result of this dependency. If a maintainer wishes to replace c_2 with another compatible component, then c_1 must also be replaced since it depends specifically on c_2. As a result, *components* with fixed dependencies on other implementations, like `A1`, `B1`, and `C1` in Figure 2.8, do not support component-level maintenance. In the following section we address this problem.

2.4.2 Deferred Dependencies

As we discussed in Section 2.1.3, in order to foster substitutability, components must be designed and implemented to *conform* to behavioral interfaces and also to *require* use of any components that conform to those interfaces. Consider again the component relationships depicted in Figure 2.2. For now, assume that the element type of list, stack, and queue has been fixed, say to type `Integer`. (We will consider the more general case without this assumption later in this section.) If we model the `(Integer)` list implementations on the bottom of the figure as components in CI and the `(Integer)` list interface in the center as a specification in AI, then the "conforms to" relationship shown may be modeled by the **imps** relation defined in Equation 2.2. The issue we address in this section is how to model the "requires an implementation of" relationships shown on the top of Figure 2.2.

It would be convenient to use elements of CI as the models for all implementation components. However, implementations with dependencies expressed in terms of a behavioral interface specification are different in kind from implementations that have fixed dependencies. To understand the difference, consider the five components shown in Figure 2.9. Assume that the two list implementations at the bottom of the figure, `IL1` and `IL2`, are implementations in CI with no external dependencies. That is, they only use operations and types provided directly by L (including, in this example, type `Integer`). Assume that the behavioral interface depicted in the center of the figure, `IL`, is a specification in AI and that both list implementations conform to this specification as indicated by the arrows labeled **imps**. The stack implementation depicted in the upper left corner of the figure, `IS1`, directly uses list implementation `IL1` as shown. We may model `IS1` as a component in CI with a corresponding behavior $\mathcal{S}$(`IS1`) defined in terms of $\mathcal{S}$(`IL1`).

The component labeled `IST1` in the upper right corner of Figure 2.9 is an `Integer` stack implementation that may use *any* list implementation that conforms to `IL`. We say that `IST1` has a *deferred dependency* on an implementation of `IL` or that it "needs" an implementation of `IL`. (We define the **needs** relation below.) Assume that the content of `IST1` is identical to that of `IS1` except that where `IS1` names list

[7]Since the set of *all* concrete instances CI is fixed in our model, "making a change to c_1" is equivalent to shifting attention to another component in CI that may or may not depend on c_2.

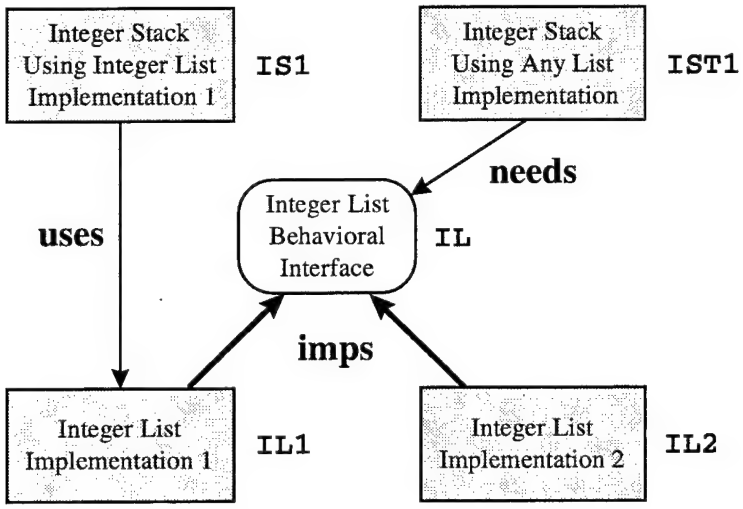


Figure 2.9: Fixed and Deferred Dependencies

operations specifically provided by IL1, IST1 refers to list operations as specified in IL. (We will examine the details of how a component with deferred dependencies may be encoded in specific programming languages in Chapters 4 and 5.) The problem with modeling IST1 as a concrete instance is that this component does not describe a single component behavior in B. IST1 characterizes a set with as many different stack behaviors as there are list behaviors in $\mathcal{S}(\texttt{IL})$. If we select IL1 to satisfy IST1's need for an implementation of IL, then we expect the resulting behavior to be the same as that described by IS1. If we select IL2 instead, then the resulting behavior may be different.

We model implementation components with deferred dependencies as members of the set CT of concrete templates introduced in Section 2.2.1. A component with a deferred dependency may be viewed as a template for generating concrete instances. We model the *meaning* of a concrete template as a function from one behavior in B to another behavior in B. The semantic function $\mathcal{S}$ (restricted to the domain CT) maps each concrete template $t \in CT$ to a function in the set of all functions from B to B. The domain of the function $\mathcal{S}(t)$ is the subset of B defined by the specification $a \in AI$ used to express the deferred dependency of t. The range of $\mathcal{S}(t)$ is the subset of B that includes the behaviors corresponding to all concrete instances that may be generated by instantiating t with any concrete instance that implements a.

The relation **needs** : $CT \times AI$ models a deferred dependency between a concrete template and an abstract instance. The relation is defined as follows:

$$\textbf{needs}(t, a) \equiv \text{domain}(\mathcal{S}(t)) = \mathcal{S}(a) \tag{2.6}$$

The predicate **needs**(t, a) may be read as "concrete template t needs an implementation (any implementation) of abstract instance a". For any $c \in CI$ for which **imps**(c, a) holds, the result of instantiating t with c is a concrete instance c' for which **uses**(c', c) holds. That is, the concrete instance generated by the instantiation uses the concrete instance that was chosen to instantiate the concrete template.

Figure 2.10 shows how the components and relationships in Figure 2.9 (except for IL2) are modeled. IST1, the stack implementation that needs any concrete instance that implements IL, is modeled as a concrete template in CT. IST1's deferred dependency on an implementation of IL is indicated by the arrow labeled **needs** from IST1 in CT to IL in AI. The semantic function $\mathcal{S}$ maps IST1 to the element $\mathcal{S}$(IST1) in the set of functions from B to B. Since IST1 needs an implementation of IL, the domain of the function $\mathcal{S}$(IST1) is $\mathcal{S}$(IL) which is depicted by the dashed oval within B. Since the list implementation IL1 implements the list specification IL, IL1 may be used to fulfill IST1's requirement. Figure 2.10 conveys this on the left side with the **needs** and **imps** relationships and on the right side by showing the behavior $\mathcal{S}$(IL1) in the subset $\mathcal{S}$(IL) of B. When the function $\mathcal{S}$(IST1) is applied to the behavior $\mathcal{S}$(IL1) the result is the behavior $\mathcal{S}$(IS1) as depicted by the dashed line from $\mathcal{S}$(IL1) to $\mathcal{S}$(IS1). Thus the behavior implemented by the concrete template IST1 instantiated with the concrete instance IL1 is the same as the behavior implemented by IS1 with its fixed dependency on IL1.

The final aspect of the model is the meaning of abstract templates. We model the meaning of an abstract template as a function from the set B of behaviors to the set $\mathcal{P}(B)$, the power set of behaviors. For an abstract template $u \in AT$ and a concrete instance $c \in CI$, there is an abstract instance $a \in AI$ such that $\mathcal{S}(u)(\mathcal{S}(c)) = \mathcal{S}(a)$. Thus the meaning of an abstract template may be viewed as a function that, when applied to the meaning of an implementation, produces the meaning of a specification. This is a somewhat different model of abstract templates than that defined by ACTI[8]. Nevertheless, this view of abstract templates is sufficient for our needs as presented in Chapter 3.

Figure 2.11 shows all of the spaces defined within the model including the set of abstract templates AT and the set of mathematical theory modules M. In this figure, the abstract template LT is a specification of a list just like IL, except that it is parameterized by the type of item contained in the list. That is, LT specifies a list template rather than an integer list. The components I and I1 represent an integer type specification and implementation, respectively. (Although these are not depicted in Figure 2.11, IS1 **uses** I1, IL1 **uses** I1, IST1 **uses** I1, and IL **uses** I1.) In this situation, the behavior specified by instantiating LT with I1 is the same as that

[8]In the ACTI model, an abstract template is a function from an *abstract instance* to another abstract instance. [Edw95, §4.8]

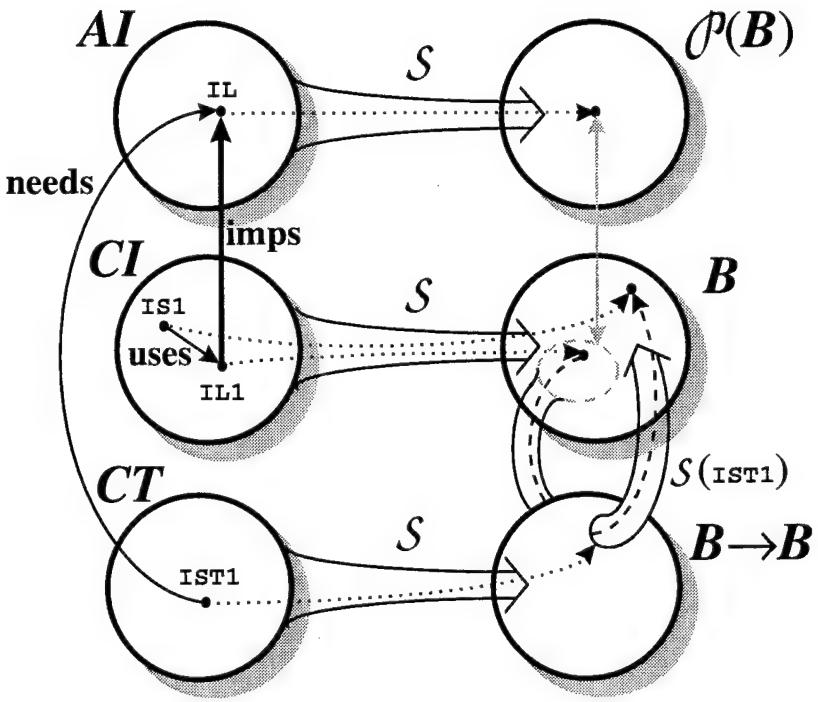


Figure 2.10: Concrete Templates And The Needs Relation

imps	$\mathbf{imps}(c, a)$	$\equiv$	$\mathcal{S}(c) \in \mathcal{S}(a)$
exts	$\mathbf{exts}(a_2, a_1)$	$\equiv$	$\mathcal{S}(a_2) \subseteq \mathcal{S}(a_1)$
uses	$\mathbf{uses}(c_1, c_2)$	$\equiv$	$c_1 = c_2 \vee$ $c_2 \in \mathbf{cxt}(c_1) \vee$ $\exists c \in \mathbf{cxt}(c_1) \mid \mathbf{uses}(c, c_2)$
needs	$\mathbf{needs}(t, a)$	$\equiv$	$\text{domain}(\mathcal{S}(t)) = \mathcal{S}(a)$

Table 2.1: Summary of Modeled Component Relations

specified by IL. This instantiation of LT is depicted in Figure 2.11 as the function application arrow traveling from $\mathcal{S}(\texttt{I1})$ in $\mathcal{B}$ through the function $\mathcal{S}(\texttt{LT})$ to $\mathcal{S}(\texttt{IL})$ in $\mathcal{P}(\mathcal{B})$.

Figure 2.11 also shows that the integer specification, abstract template I, depends on the mathematical integer theory description ITHRY. The interpretation of ITHRY, $\mathcal{I}(\texttt{ITHRY})$, is shown simply as a point in V. All of the components (strings in L) shown in the figure rely either directly or indirectly on ITHRY (these **uses** relationships are not shown to reduce clutter). Thus, each component's meaning, shown as a point or function (a set) in V, is constructed, in part, from the set $\mathcal{I}(\texttt{ITHRY})$.

2.5 Chapter Summary

In this chapter we have developed a set theoretic model of behavioral relationships between software components. The purpose of this model is to describe the behavioral relationships between software components needed to support component-level maintenance. Section 2.1 motivates the need for component-level maintenance and relationships that express behavioral conformance and behavioral requirements. Section 2.2 describes our component model, which includes abstract and concrete templates (parameterized specifications and implementations), abstract and concrete instances (non-parameterized specifications and implementations), and mathematical theory modules.

The model defines the meaning of each concrete instance as an element in the set $\mathcal{B}$ of "behaviors". The meaning of an abstract instance is defined as a set of behaviors, a member of the power set of $\mathcal{B}$, $\mathcal{PB}$. The meaning of a concrete instance is defined as a function from $\mathcal{B}$ to $\mathcal{B}$. The meaning of an abstract template is defined as a function from $\mathcal{B}$ to $\mathcal{PB}$.

In Section 2.3, we defined **imps**, a conformance relationship between concrete and abstract instances, and **exts**, a conformance relationship between two abstract

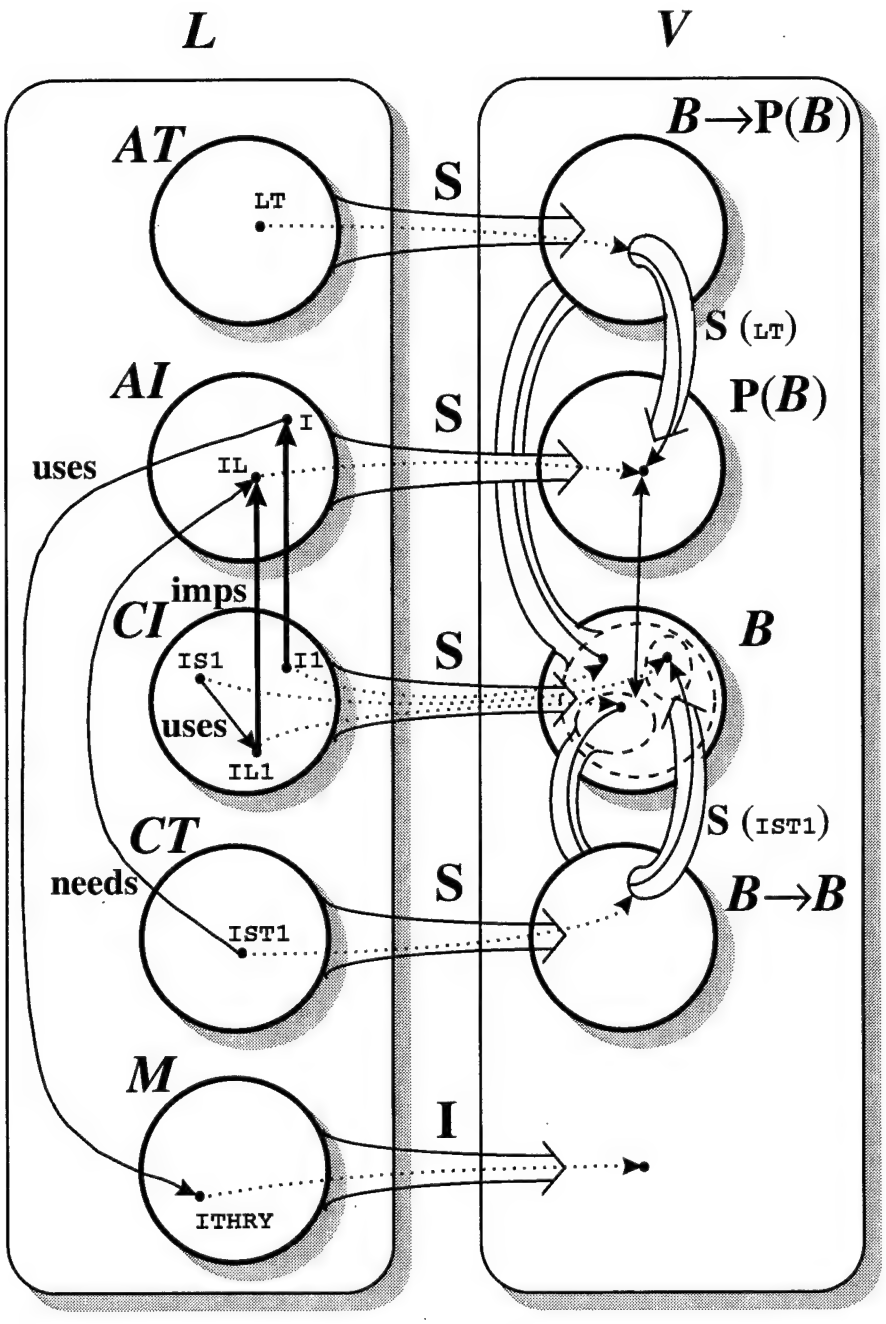


Figure 2.11: The Big Picture

instances. In Section 2.4 we defined the fixed dependency relation **uses** and the deferred dependency relation **needs**. Table 2.1 lists these relations and their definitions.

CHAPTER 3

A USEFUL SET OF SOFTWARE COMPONENT RELATIONSHIPS

In this chapter we define a useful set of software component relationships based on the model of components and relations developed in Chapter 2. We do not claim that these are the only useful relationships between components nor do we claim that they are ideally suited to all approaches to software development. However, as we will explain, the relationships presented in this chapter are well-suited for building and maintaining component-based software systems. In this chapter we also introduce the graphical notation of component coupling diagrams used to depict each relationship described.

This chapter presents a sequence of simple example components to demonstrate the relationships described. Section 3.1 introduces a specification and implementation notation used to encode example components. Section 3.2 describes the dependency relationship **uses**, corresponds directly to the **uses** relation described in Section 2.4.1. Section 3.3 describes the behavioral relationship **implements** which is based on the **imps** relation described in Section 2.3.1. Section 3.4 describes the deferred dependency relationship **needs** which is based on the **needs** relation described in Section 2.4.2. Section 3.5 describes the behavioral relationship **extends** which is based on the **exts** relation described in Section 2.3.2. In Section 3.7 we provide a summary of the relationships defined in this chapter.

3.1 Component Notation

The component relationships described in this chapter are language-independent in the sense that they are not tied to specific language mechanisms. The relationships reflect design-level information that may be encoded in various ways with various implementation and specification languages. Nevertheless, some languages provide much better support than others for encoding these relationships. Chapter 4 discusses language support for encoding these relationships.

In this chapter we present examples of components encoded in a custom notation with specification elements similar to those of the RESOLVE language and implementation elements similar to those of Ada. The notation has been simplified (with respect to RESOLVE and Ada) in order to shorten example code and minimize the need to address details not directly relevant to the issues discussed in this chapter. Some of the simplifications involve name space control (use of unqualified versus qualified identifier names), operation parameter mode specification, features built into the language, implementation-level encapsulation, object initialization and finalization, and minor syntactic details (such as the use of semicolons). The components encoded in RESOLVE/Ada95 shown and discussed in Chapter 5 address these and other issues with specific solutions based on the capabilities and limitations of RESOLVE and Ada.

As in Chapter 2, we use the term "component" to refer to a software module that describes either an implementation of behavior or a specification of behavior. Furthermore the description of behavior may be parameterized (a template component) or not (an instance component). We continue to use the terms "concrete instance", "concrete template", "abstract instance", and "abstract template" to refer to the four kinds of components which result from this categorization. In the examples, we will prefix each component name with the string "`CI_`", "`CT_`", "`AI_`", or "`AT_`" to indicate its classification as one of these four kinds of components. Note that this is a naming convention only and not part of the notation syntax.

In the notation used in this chapter, both specification components (abstract instances and abstract templates) and implementation components (concrete instances and concrete templates) are encoded with the same basic format. Each component has: a header, an optional `context` section, an optional `auxiliary` section, an `interface` section, and a terminal `end` delimiter. The header begins with either `specification` or `implementation` followed by the component name which optionally may be followed by `extends` clauses in specifications and `implements` clauses in implementations. The `context` section lists all direct fixed dependencies encoded with `uses` clauses followed by all deferred dependencies encoded with `needs` clauses. The deferred dependencies constitute the component parameter section. If a component has no external dependencies — it is constructed solely in terms of elements built into the language — then its empty context section may be omitted.

The `auxiliary` section in a specification component may include definitions that describe the behavior specified in the `interface` section. In addition to specification-only definitions, the `auxiliary` section of an implementation component also may include *local* definitions of program types, operations, and variables used to describe the behavior implemented in the `interface` section. That is, any program types, operations, and variables defined in the `auxiliary` section may be referenced only within the remainder of that section and in the following `interface` section. The `auxiliary` section may be omitted if it is empty. The `interface` section provides

```
specification AI_Flipflop

   interface

      type Flipflop is modeled by BOOLEAN
         exemplar  ff
         initially ff = FALSE

      procedure Toggle (f : Flipflop)
         ensures f = NOT #f

      function Test (f : Flipflop) : Boolean
         ensures  Test = f

end AI_Flipflop
```

Figure 3.1: Abstract Instance `AI_Flipflop`

a specification or implementation of program behavior in terms of program types and operations. Program type, operation, and variable definitions in the `interface` section may be made available for use by other components.

Figure 3.1 shows a very simple abstract instance named `AI_Flipflop`. This abstract component provides a model-based specification [Win90] for a two-state device, a flip-flop, for which the current state may be toggled and tested (queried). The `context` and `auxiliary` sections are not shown since this component only uses built-in types and operations. The `interface` section includes definitions of the abstract data type `Flipflop` and two associated operations `Toggle` and `Test`.

The examples in this chapter assume that all components have visibility over the components `CI_Boolean_1` and `CI_Integer_1` that define the scalar *program types* `Boolean` and `Integer`. These types and their associated operations (e.g., `or` for `Boolean` and `+` for `Integer`) may be used without any reference in the context section to the components in which they are defined. Thus `Boolean` and `Integer` may be considered as built-in program types of the language. As in most programming languages, program types are used to ensure that program variables are used in legal contexts.

The program types `Boolean` and `Integer` are modeled by the *math types* `BOOLEAN` and `INTEGER`, respectively. The math types `BOOLEAN` and `INTEGER` are specified in the *math theory modules* `MI_Boolean_Theory` and `MI_Integer_Theory`. These two math theory modules are also built-in theories of the language in the sense that the math types and math operations that they define (e.g, `OR` for `BOOLEAN` and `+` for `INTEGER`) may be referenced without mentioning their defining math theory modules in the

context section. Math theory modules provide formal specification of mathematical theories that may be used to mathematically model program behavior. They serve the same role as math modules in RESOLVE [WOZ91] and as *traits* in the Larch Shared Language [GHW85]. Note that we do not consider math modules to be software "components" since they do not describe program behavior and their use in no way effects the operational behavior of component-based systems. To help distinguish between program and math types and operations, we use all upper case identifiers for math types and operations and mixed case identifiers for program types and operations. Common operator symbols such as "+" and "=" are exceptions and may be distinguished by context.

In the `interface` section of `AI_Flipflop` (Figure 3.1) program type `Flipflop` is declared by defining its mathematical model to be the math type `BOOLEAN`. Thus, the *abstract* state space used to model a flip-flop is the set {`FALSE`, `TRUE`}. The `exemplar` clause states that the identifier `ff` will represent a prototypical object for specifying properties of all objects (values of variables) of type `Flipflop`. The `initially` clause states that the initial abstract state of a `Flipflop` object is `FALSE`.

The operation `Toggle` is specified using a relational `procedure` signature that takes a `Flipflop` object as its single argument. Execution of a procedure may change the abstract values of all of the operation's arguments. The pre-condition of `Toggle`, expressed by a `requires` clause, is not shown since it places no restrictions on the states from which `Toggle` may be meaningfully invoked (a flip-flop may be toggled from either state). The post-condition of `Toggle`, expressed by the `ensures` clause, specifies that after execution of `Toggle` the abstract state corresponding to the concrete state of the argument is the negation of the abstract state prior to execution. In `requires` and `ensures` clauses, formal parameter identifiers (`f` in this example) denote objects of the math type used to model the parameter's program type. In an `ensures` clause, an argument prefixed by "#" denotes the value of the argument *prior* to execution of the operation being specified. Thus `Toggle` changes the flip-flop from one state to the other state.

The operation `Test` is specified using a `function` signature that takes a `Flipflop` object as its single argument and returns a value of the *concrete* program type `Boolean` (defined by the concrete instance `CI_Boolean_1`). Execution of a function may not change the *abstract* value of any of the operation's arguments. The assertion that function argument values do not change is an implicit conjunct of a function's `ensures` clause. Like `Toggle`, `Test` has no pre-condition. The `ensures` clause of `Test` specifies that the value returned by `Test` (denoted by `Test`) corresponds to the abstract value of `Test`'s argument. Thus `Test` may be used to query the state of the flip-flop. As a behavioral interface specification, `AI_Flipflop` requires all conforming implementations to provide (at least) a representation for type `Flipflop` (initialized to a value representing `FALSE`), an implementation for `Toggle`, and an implementation for `Test`.

```
implementation CI_Flipflop_2

   interface

      type Flipflop is represented by
            state : Integer range 0 .. 255 := 0
         end representation

      procedure Toggle (f : Flipflop) is
         begin
            f.state := (f.state + 1) mod 256
         end Toggle

      function Test (f : Flipflop) : Boolean is
         begin
            return ((f.state mod 2) = 1)
         end Test

end CI_Flipflop_2
```

Figure 3.2: Concrete Instance `CI_Flipflop_2`

Figure 3.2 shows a simple concrete component named `CI_Flipflop_2`. This component provides one of infinitely many possible implementations of the flip-flop ADT specified by `AI_Flipflop`. By convention we will suffix implementation component names with an underscore followed by a number used to distinguish between different implementations of the same specification. For example, the name "`CI_Flipflop_2`" may be interpreted as the *second* concrete instance implementing `AI_Flipflop`. (Assume that `CI_Flipflop_1`, not shown, is the obvious implementation using a `Boolean` for the representation of `Flipflop`.) The structure of `CI_Flipflop_2` is very similar to that of `AI_Flipflop` shown in Figure 3.1. As with `AI_Flipflop`, the `context` and `auxiliary` sections are empty and not shown since only built-in types and operations are used within the component. The `interface` section includes a definition of the *concrete* program type `Flipflop` and implementations of the two associated operations `Toggle` and `Test`.

The data representation of type `Flipflop` in `CI_Flipflop_2` consists of a single representation component, labeled `state`. The `state` component is an object of the concrete type `Integer` (as defined in `CI_Integer_1`) restricted to the interval `[0, 255]` and having an initial value of `0`. The operation `Toggle` increments the value of its argument's `state` component by one each time it is called unless the value is `255`

in which case it is reset to 0. The := (assignment)[9], + (addition), and mod (modulus) operations are provided by CI_Integer_1. The operation Test returns the Boolean value True if the value of its argument's state component is odd, otherwise it returns False.

3.2 The uses Relationship

The **uses** relationship describes the fixed syntactic dependency of one component on another. The **uses** relationship may be defined informally as follows:

> Component C_1 **uses** component C_2 if and only if the meaning of C_1 depends either directly or indirectly on the meaning of C_2.

The **uses** relationship is modeled by the **uses** relation defined in Equation 2.5 in Chapter 2. If component C_1 **uses** component C_2 *directly*, then C_2 is in the context of C_1 and entities defined in C_2 may be used in the definition of C_1. If component C_1 **uses** component C_2 *indirectly*, then C_2 is not in the context of C_1, but is in the context of some component that C_1 **uses**. In both cases, the behavior either specified or implemented by C_1 depends on (is defined in terms of) the behavior specified or implemented by C_2.

The **uses** relationship is very important from a maintenance perspective. If component C_1 **uses** component C_2, then any modification to C_2 may alter the behavior described by C_1. Also, in order to fully understand a component, it may be necessary to understand aspects of all other components that it **uses**. The **uses** relationship is often viewed as a "client/supplier" relationship [Mey88, p. 73] [Boo91, p. 101]. If C_1 **uses** C_2 then C_1 is a *client* of C_2 which is a *supplier* to C_1. The **uses** relationship gives C_1 *visibility* to elements defined in C_2 and elements defined in components that C_2 **uses**. Depending on the language mechanisms used, C_1 may or may not have visibility to all features defined by C_2 and the components it **uses**.

The most familiar form of **uses** describes coupling between two concrete instances. This relationship should be familiar to anyone who has developed software using a programming language that supports separately compiled modules. These languages have import or inclusion mechanisms that encode a direct **uses** relationship between two modules. Examples include the `with` context clause in Ada, the `import` statement in Modula-2, and the `#include` preprocessor directive in C++. In the notation presented in this chapter, a direct **uses** relationship is encoded with a `uses` clause[10] in the `context` section. For example, a concrete instance built specifically using CI_Flipflop_2 would include in its context section the clause "`uses CI_Flipflop_2`".

[9]Here we consider := as an operation defined on type Integer, not as a program statement as in Ada.

[10]Note that in Ada, the `use` clause serves the different purpose of allowing identifiers already in scope to be referenced without using their fully qualified names.

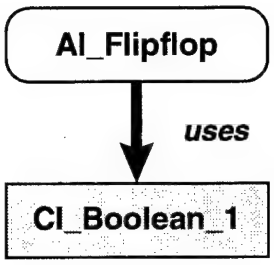


Figure 3.3: The **uses** Relationship

The **uses** relationship also may describe a fixed dependency of one specification on another specification, of an implementation on a specification, or of a specification on an implementation. Figure 3.3 shows a component coupling diagram (CCD) that graphically depicts the (implicit) **uses** relationship between `AI_Flipflop` and `CI_Boolean_1`. In CCD's, abstract components (both instances and templates) are depicted as clear boxes with rounded corners. Concrete components (instances and templates) are depicted as shaded rectangular boxes. The component name is shown within each box. The thick solid arrow from `AI_Flipflop` to `CI_Boolean_1` represents the **uses** relationship between the two components. We use thick arrows to depict dependency (coupling) relationships. Note that typically we will *not* show implicit dependencies on built-in components in CCD's as shown in this example.

3.3 The implements Relationship

The **implements** relationship is a behavioral relationship between a concrete component and an abstract component. The **implements** relationship may be defined informally as follows:

> Concrete component C **implements** abstract component A if and only if C provides an implementation of all behavior specified by A.

The **implements** relationship is a conformance relationship between C and A modeled by the **imps** relation defined in Equation 2.2. However, **imps** describes a relationship between instance components only. We extend the definition of **implements** to include implementations that are templates. In this extended view, **implements** is an overloaded term for three distinct relationships:

- If C is a concrete instance and A is an abstract instance, then the claim that C **implements** A is an assertion that **imps**(C, A) holds.

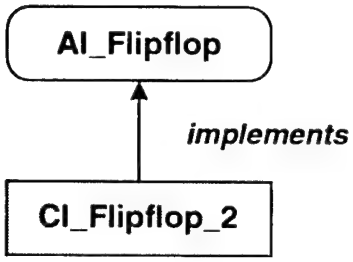


Figure 3.4: The **implements** Relationship

- If C is a concrete template and A is an abstract instance, then the claim that C **implements** A is an assertion that for *any* instantiation C'' of C, **imps**(C'', A) holds.
- If C is a concrete template and A is an abstract template, then the claim that C **implements** A is an assertion that for *any* instantiation C'' of C there exists *some* instantiation A' of A, such that **imps**(C'', A') holds.

We will discuss examples of each of these three cases in this chapter. In this section we discuss the first case, the **implements** relationship between a concrete instance and an abstract instance. In Section 3.4 we present examples of the other two cases.

As an example of the **implements** relationship, consider the abstract instance `AI_Flipflop` shown in Figure 3.1 and the concrete instance `CI_Flipflop_2` shown in Figure 3.2. We *claim* that `CI_Flipflop_2` **implements** `AI_Flipflop`. That is, anywhere that there is a requirement for the behavior specified by `AI_Flipflop`, the implementation `CI_Flipflop_2` may be used to satisfy that requirement. This relationship is graphically depicted in the CCD in Figure 3.4. The thin solid arrow from `CI_Flipflop_2` to `AI_Flipflop` represents the **implements** relationship between the two components. We use thin arrows to depict behavioral relationships which are **not** dependency (coupling) relationships.

Aside from its suggestive name and obvious structural similarity, the content of `CI_Flipflop_2` as shown in Figure 3.2 contains no statement of its purported relationship to `AI_Flipflop`. If we assume that `CI_Flipflop_2` was specifically designed to implement `AI_Flipflop`, then it seems natural that this information should be recorded in the source code of `CI_Flipflop_2` to help explain the intended behavior of the component to prospective maintainers. Furthermore, information explaining *how* `CI_Flipflop_2` may be viewed as implementing `AI_Flipflop` would also be useful to anyone attempting to justify the claim that the **implements** relationship really does hold between these two components.

```
implementation CI_Flipflop_3
   implements  AI_Flipflop

   interface

      type Flipflop is represented by
            state : Integer range 0 .. 255 := 0
         end representation
         exemplar ff_rep
         correspondence ff = ((ff_rep.state MOD 2) = 1)

      procedure Toggle (f : Flipflop) is
         begin
            f.state := (f.state + 1) mod 256
         end Toggle

      function Test (f : Flipflop) : Boolean is
         begin
            return ((f.state mod 2) = 1)
         end Test

end CI_Flipflop_3
```

Figure 3.5: Concrete Instance CI_Flipflop_3

Consider the concrete instance CI_Flipflop_3 shown in Figure 3.5. The only differences between CI_Flipflop_2 and CI_Flipflop_3 are the implements clause in the header and the exemplar and correspondence clauses in interface section of CI_Flipflop_3. None of these three additions affects the operational behavior described by CI_Flipflop_3. Thus CI_Flipflop_3 describes an implementation of behavior identical to behavior implemented by CI_Flipflop_2.

The implements clause in the header of CI_Flipflop_3 explicitly records an *intended* **implements** relationship with AI_Flipflop. The exemplar clause and correspondence clause in CI_Flipflop_3 explain an intended correspondence between the *representation states*, modeled by INTEGER, and the *abstract states*, modeled by BOOLEAN. The identifier MOD names a math operation for integer modulus (MOD is defined in MI_Integer_Theory). The name f_rep.state denotes the abstract value corresponding to the representation-level value of the state field of an object of type Flipflop. Since the state field is an object of program type Integer, its values correspond to abstract values of math type INTEGER. The correspondence defines the relation {<0,FALSE>, <1,TRUE>, ..., <254,FALSE>, <255,TRUE>} mapping even representation values to the abstract state FALSE and odd values to TRUE.

The additional information encoded in `CI_Flipflop_3` serves several useful purposes. The statement that `CI_Flipflop_3` **implements** `AI_Flipflop` "officially" records design intent of the implementer of `CI_Flipflop_3` and tells a potential maintainer (or a library browsing tool) where to look for a description of the requirements this component must satisfy. The `implements` clause also makes explicit the obligation that `CI_Flipflop_3` must conform structurally and semantically to `AI_Flipflop` when used as an implementation of `AI_Flipflop`. For example, the `implements` clause shown in Figure 3.5 might require a compiler to check for the structural conformance of `CI_Flipflop_3` to `AI_Flipflop` when `CI_Flipflop_3` is compiled. Both `CI_Flipflop_2` and `CI_Flipflop_3` conform structurally to `AI_Flipflop` since they provide a representation for the type `Flipflop` and operations with names and parameter profiles that match those of `AI_Flipflop`.

Within a library of "certified" components, the `implements` clause (perhaps in object code form) also might be interpreted as a statement that the **implements** relationship has been justified to whatever degree required. In this case, only concrete components that have been certified to conform structurally *and* semantically to the specifications which they claim to implement would be entered into the library. Officially recording justified `implements` relationships elsewhere, however, provides a more flexible solution. For example, if justified `implements` (and `extends`) relationships are recorded in a component library database, new relationships may be added and existing ones "revoked" without modifying any component content. Such a database of relationships would be useful for component library browsing as well as for use by component integration tools.

The claim that a concrete component **implements** an abstract component is an assertion that the concrete component is a *correct* implementation with respect to the specification provided by the abstract component. The `correspondence` clause provides information necessary for formally verifying the correctness of an ADT and thus for justifying that the **implements** relationship holds between a component that implements a type and a component that specifies the type. The relation expressed by the `correspondence` clause is also called an *abstraction function*[LG86, p 70], a *retrieve function*[Jon90, p 182], and more generally an *abstraction relation*[SW097].

In short, the `correspondence` clause provides a way to compare the effect of executing operations on the concrete representation state described by an implementation component, with the effect of executing the same operations on the abstract state described by a specification component. In order for an implementation to be considered correct, any concrete state reachable from a legal sequence of (zero or more) operations must correspond to an abstract state reachable from the same sequence of operations. Note that the `initially` clause (as shown in Figure 3.1) ensures that the concrete state of an object corresponds to its specified abstract state prior to execution of any operations that affect the state of that object. This serves as a basis for an induction argument stating that after an arbitrarily long sequence

of operations, the concrete state of an object will still correspond to an appropriate abstract state as specified. The details of formally justifying the **implements** relationship are beyond the scope of this dissertation and have been discussed elsewhere in terms of formal verification of ADT's [Hoa72, LG86, Jon90, EHO94].

`AI_Flipflop`, `CI_Flipflop_2`, and `CI_Flipflop_3` are so simple that the claimed **implements** relationships appear easily justified in both cases. But useful software components tend to be much more complex than these simple examples. In general, the process of convincingly justifying that a concrete component **implements** an abstract component may require a great deal of effort and even creativity. By explicitly stating the correspondence between an implementation's data representation and a model used to specify desired program behavior, the component implementer documents a critical aspect of how the behavior described by an implementation may be viewed as corresponding to the behavior described by a specification.

A component implementer may use the **implements** relationship to state how an implementation component, possibly in object code form, should be viewed by prospective clients. By claiming that concrete component C **implements** abstract component A, the implementer is claiming that A serves as an appropriate simplified description or "cover story" for behavior implemented by C. With `CI_Flipflop_3`, the abstract state space modeled by `BOOLEAN` is different and simpler (much smaller) than the concrete state space of `INTEGER` modulo 256. In this case, `AI_Flipflop` presents a simpler conceptual view or "mental model" of flip-flop behavior than that described by `CI_Flipflop_3`. In addition to supporting substitutability, a primary goal of establishing the **implements** relationship is to identify an abstract description of an implementation's behavior that is easier for a client to understand than the description of behavior provided by the implementation.

The benefits of including the `implements`, `exemplar`, and `correspondence` clauses in a concrete component should be clear. Nevertheless, there are reasons why it might be useful to maintain this information elsewhere, either in addition to, or perhaps even instead of maintaining it within the content of concrete components. We discuss this issue in Section 4.3.5. For now, we reiterate that the **implements** relationship to `AI_Flipflop` may be justified for both `CI_Flipflop_2` and `CI_Flipflop_3` and that the additional information provided by `CI_Flipflop_3` serves to explain how this **implements** relationship may be justified.

3.4 The needs Relationship

The **needs** relationship is a dependency relationship between a concrete template and an abstract instance. It expresses a deferred dependency between an instantiation of the concrete template and an implementation of the abstract instance. The **needs** relationship may be defined informally as follows:

> Concrete template C **needs** abstract instance A if and only if C **uses** A and, for all instantiations of C, C's references to program elements in A are replaced by references to the corresponding program elements in some concrete instance that **implements** A.

The relationship name "**needs**" is short for "needs an implementation of". The **needs** relation (defined in Equation 2.6) models the **needs** relationship between two components. However, in this chapter we allow a concrete template to have more than one deferred dependency. That is, a single concrete template may need implementations of more than one abstract instance. Furthermore, it is possible that a concrete template may need more than one implementation of the *same* abstract instance.

3.4.1 Implementation-Level needs

Before looking at an example of a deferred dependency expressed by the **needs** relationship, we will examine a closely related fixed dependency relationship. To set up both examples, we introduce a new abstract instance. Figure 3.6 shows the abstract instance `AI_Threeway` which serves as an austere interface describing the behavior of a "three-way" light bulb switch with one "off" state and three different "on" states. The `auxiliary` section includes the declaration of `Z4`, a math subtype of `INTEGER` constrained to the interval [0, 3]. The `interface` section of `AI_Threeway` specifies the type `Threeway` and the operations `Advance` and `On`. The type `Threeway` is modeled by `Z4`. The design intent here is that the abstract state `0` models the switch's "off" state and that the states `1`, `2`, and `3` model the "low", "medium", and "high" brightness levels respectively. The `Advance` operation changes the state of its argument to the next higher brightness level or to "off" from "high". The `On` operation returns `True` if the switch is in one of the three "on" states corresponding to `1`, `2`, or `3`. If necessary, a client could cycle through the switch states using `Advance` and `On` to determine the brightness level.

Figure 3.7 shows the concrete instance `CI_Threeway_1`. `CI_Threeway_1` **implements** `AI_Threeway` and **uses** `CI_Flipflop_3` (shown in Figure 3.5) to do so. The fixed dependency on `CI_Flipflop_3` is expressed with a `uses` clause in the `context` section. `CI_Threeway_1` also **uses** `AI_Threeway` in order to express the correspondence. Since `CI_Threeway_1` **uses** `CI_Flipflop_3`, it has direct access to the representation of type `Flipflop` defined by `CI_Flipflop_3`. From the perspective of `CI_Threeway_1`, the concrete type `Flipflop` is modeled by a singleton of math type `INTEGER` (the model of concrete type `Integer`) constrained to the interval `[0, 255]`. Thus by referring to a specific concrete component, a client component such as `CI_Threeway_1` commits to a specific concrete representation.

The `auxiliary` section includes the definition of the math operation `PARITY` used in the `correspondence` clause. The `interface` section defines the representation of

```
specification AI_Threeway

   auxiliary

      math subtype Z4 is INTEGER
          exemplar Z
          constraint 0 <= Z and Z <= 3

   interface

      type Threeway is modeled by Z4
         exemplar T
         initially T = 0

      procedure Advance (t : Threeway)
         ensures  t = (#t + 1) MOD 4

      function On (t : Threeway) : Boolean
         ensures  On = (t > 0)

end AI_Threeway
```

Figure 3.6: Abstract Instance AI_Threeway

type Threeway and implementations for Advance and On. The representation of type Threeway is a pair of objects, labeled ff1 and ff2, both of type Flipflop as defined in CI_Flipflop_3. These two objects maintain which of the four states the switch is in as explained by the correspondence. The two flip-flops are simply used as a two-bit counter. Since CI_Flipflop_3 represents Flipflop with an object of type Integer constrained to the interval [0, 255], CI_Threeway_1 represents Threeway as a pair of objects of type Integer. Thus the *representation-level* model of type Threeway as defined by CI_Threeway_1 is a pair of INTEGER objects with values constrained to the interval [0, 255].

The correspondence clause defines a relation mapping each of the 256^2 representation states to one of the four abstract states of Threeway defined in AI_Threeway. For example, any representation in which both flip-flop objects have an even value, maps to the abstract state 0 which models the switch in the "off" position.

The implementation of the Advance operation is defined in terms of Toggle and Test. The implementation of On is defined in terms of just Test. These implementations do not *directly* access the representation-level state of the flip-flops they manipulate. Thus in this case, there is no reason why CI_Threeway_1 needs to be designed with a fixed dependency on CI_Flipflop_3. Nevertheless, without reference to an abstract specification of CI_Flipflop_3 (at the point in CI_Threeway_1 where

```
implementation CI_Threeway_1
   implements  AI_Threeway

   context
      uses CI_Flipflop_3

   auxiliary
      math operation PARITY (I : INTEGER) : INTEGER
         definition  I MOD 2

   interface

      type Threeway is represented by
            ff1 : Flipflop
            ff2 : Flipflop
         end representation
         exemplar tw_rep
         correspondence  tw = 1 * PARITY(tw_rep.ff1.state) +
                              2 * PARITY(tw_rep.ff2.state)

      procedure Advance (t : Threeway) is
         begin
            Toggle (t.ff1)
            if not Test(t.ff1) then
               Toggle (t.ff2)
            end if
         end Advance

      function On (t : Threeway) : Boolean is
         begin
            return(Test(t.ff1) or Test(t.ff2))
         end On

end CI_Threeway_1
```

Figure 3.7: Concrete Instance CI_Threeway_1

CI_Flipflop_3 is used), the correspondence must be defined in terms of the Flipflop representation provided by CI_Flipflop_3.

Figure 3.8 shows concrete *template* CT_Threeway_1, an alternative implementation of AI_Threeway that presents an example of the **needs** relationship. Instead of depending on a specific flip-flop implementation, CT_Threeway_1 is parameterized by a concrete instance that **implements** AI_Flipflop. The deferred dependency of CT_Threeway_1 on *some* (*any*) implementation of AI_Flipflop is expressed by the needs clause at the end of the **context** section. Any component that includes one or more needs clauses in its context section is a template. The identifier CI_Flipflop

```
implementation CT_Threeway_1
   implements  AI_Threeway

   context
      uses  AI_Flipflop
      needs CI_Flipflop implementing AI_Flipflop

   auxiliary
      math operation BTI (B : BOOLEAN) : INTEGER
         definition  if B then 1 else 0

   interface

      type Threeway is represented by
            ff1 : Flipflop
            ff2 : Flipflop
         end representation
         exemplar tw_rep
         correspondence  tw = 1 * BTI(tw_rep.ff1) + 2 * BTI(tw_rep.ff2)

      procedure Advance (t : Threeway) is
         begin
            Toggle (t.ff1)
            if not Test(t.ff1) then
               Toggle (t.ff2)
            end if
         end Advance

      function On (t : Threeway) : Boolean is
         begin
            return(Test(t.ff1) or Test(t.ff2))
         end On

end CT_Threeway_1
```

Figure 3.8: CT_Threeway_1 needs AI_Flipflop

in the needs clause is a formal parameter name representing the concrete instance supplied as an actual parameter when CT_Threeway_1 is instantiated. Note that the needs clause presented here is very similar to the needs clause of Goguen's Library Interconnection Language (LIL) [Gog86, p. 22].

CT_Threeway_1 has a fixed dependency on AI_Flipflop as indicated by the second uses clause in the context section. CT_Threeway_1 uses AI_Flipflop to specify the behavioral requirements of any concrete instance supplied as an actual parameter. AI_Flipflop also provides the model for type Flipflop and behavioral specifications for operations Toggle and Test used in CT_Threeway_1. Therefore, no matter how

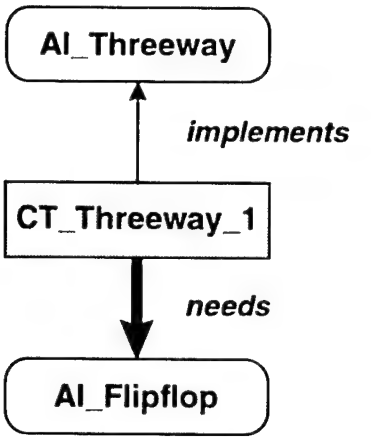


Figure 3.9: CT_Threeway_1 **needs** AI_Flipflop

CT_Threeway_1 is instantiated, it is possible to reason about the behavior of the concrete instance generated in terms of AI_Flipflop. In particular, it is possible to justify the claim that *any* instance of CT_Threeway_1 **implements** AI_Threeway which is the claim expressed in the first line of Figure 3.8. Thus this is an example of the second form of **implements** listed on page 46.

The interface section of CT_Threeway_1 is identical to that of CI_Threeway_1 except for the correspondence clause. The correspondence is different because the representation-level model of type Threeway defined in CT_Threeway_1 is different from that of type Threeway defined in CI_Threeway_1. The mathematical model of type Flipflop used in CT_Threeway_1 is BOOLEAN as specified in AI_Flipflop. Therefore the representation-level model of type Threeway defined in CT_Threeway_1 is a pair of BOOLEAN objects. The correspondence clause uses the math operation BTI (Boolean-To-Integer) to convert FALSE to 0 and TRUE to 1. The relation defined by the correspondence maps the four representation states to the four abstract states defined in AI_Threeway. For example, any representation for which values of both flip-flops correspond to TRUE maps to the abstract state 3 which models the switch in the "high" position.

Figure 3.9 shows a component coupling diagram depicting the **implements** and **needs** relationships encoded by CT_Threeway_1. The **needs** relationship is depicted by a thick solid arrow from a concrete component to an abstract component. Note that the uses relationship between CT_Threeway_1 and AI_Flipflop is not shown since a needs relationship always implies a uses relationship.

Since CT_Threeway_1 is a concrete *template*, it must be instantiated in order to describe the behavior of a concrete instance that may be integrated into a component-based system. In this case, instantiating CT_Threeway_1 requires selecting a component that **implements** AI_Flipflop to fulfill the stated need. The code describing an instantiation of CT_Threeway_1 with the CI_Flipflop_3 implementation of AI_Flipflop is as follows.

```
implementation CI_Threeway implements AI_Threeway
   by CT_Threeway_1 with (CI_Flipflop => CI_Flipflop_3)
```

The name CI_Threeway refers to the concrete instance resulting from the instantiation of CT_Threeway_1 with actual parameter CI_Flipflop_3 bound to the formal parameter CI_Flipflop. The instantiation explicitly states that CT_Threeway **implements** AI_Threeway which follows from the **implements** relationship between CT_Threeway_1 and AI_Threeway. Therefore we may reason about the behavior of the code using CI_Threeway in terms of the specification provided by AI_Threeway. In this example, the behavior implemented by CI_Threeway is identical to the behavior described by CI_Threeway_1 shown in Figure 3.7. As a result of this instantiation, only the program elements described by AI_Threeway are visible to clients of the instantiation CI_Threeway. Thus this single instantiation serves two distinct purposes. First it "fills in the holes" of CT_Threeway_1 to create a usable concrete instance, CI_Threeway. Second, it associates with this concrete instance the abstract interface AI_Threeway that describes how clients should view CI_Threeway and, in fact, the only program elements defined in CT_Threeway_1 that are available for use by client code.

Using the model developed in Chapter 2, we describe the meaning of CI_Threeway, the component described by the above instantiation, as follows. CI_Flipflop_3 and CI_Threeway are members of CI and CT_Threeway_1 is a member of CT. The behavior implemented by CI_Threeway, $\mathcal{S}$(CI_Threeway), is modeled by the element of B given by $\mathcal{S}$(CT_Threeway_1)($\mathcal{S}$(CI_Flipflop_3)).

As we noted at the beginning of this section, a concrete template may have deferred dependencies on implementations of more than one abstract component and may even have multiple independent dependencies on the same abstract component. For example, it would be possible for a different concrete template implementing AI_Threeway to have two **needs** clauses in the context, one naming CI_FF_1 and the other naming CI_FF_2, both representing concrete instances implementing AI_Flipflop. Then the two fields ff1 and ff2 representing the type Threeway could be declared of types CI_FF_1.Flipflop and CI_FF_2.Flipflop. In this case instantiation of the concrete template implementing AI_Threeway would require two actual parameters, both implementing AI_Flipflop. The actual parameters might be the same concrete instance or two different implementations of AI_Flipflop. Clearly this

simple example presents little motivation for using two different implementations of AI_Flipflop. However, Sitaraman discusses how, in general, such a strategy is useful for performance-parameterized components [Sit92].

3.4.2 Specification-Level needs

On page 45 we listed three distinct relationships to which we apply the overloaded term **implements**. The **implements** relationship between a concrete instance and an abstract instance was exemplified by CI_Flipflop_3 implementing AI_Flipflop. The **implements** relationship between a concrete *template* and an abstract instance was exemplified by CT_Threeway_1 implementing AI_Threeway. We are now ready to discuss an example of the third form of **implements**, a concrete template implementing an abstract template.

Figure 3.10 shows the encoding of abstract template AT_Stack, a behavioral interface specification for a stack which is generic with respect to the type of elements it may contain. AT_Stack has a deferred dependency on the type of elements contained in the stack. An instantiation of AT_Stack needs an implementation of a specific type to serve as the type of elements contained within the stack[11]. AT_Stack **uses** AI_AnyType, a special abstract instance, to express its requirement for a component providing a type. AI_AnyType is special in that, by convention, every concrete instance that defines at least one type **implements** AI_AnyType. AI_AnyType names a single type, AnyType, which has no associated mathematical model. When an abstract template that needs an implementation of AI_AnyType is instantiated, the mathematical model of the concrete type that corresponds to AnyType is used to describe the math model of the resulting instance. If the component serving as the actual parameter corresponding to AI_AnyType defines more than one type, then the *first* type defined is matched to AnyType. (A type other than the first type defined by a concrete instance may be matched with AnyType by using an abstract instance to "mask out" all but the desired type.)

In AT_Stack, the name AnyType represents the type supplied by the component which serves as the actual parameter corresponding to CI_StackItemType in an instantiation of AT_Stack. In the auxiliary section, the parameterized math module MT_String_Theory is instantiated to produce the math module MI_String_Theory which defines the math type STRING used to model a stack. The formal (type) parameter ItemType of MT_String_Theory represents the math type of the elements of the math type STRING. Thus, for example, if AT_Stack were instantiated with CI_Integer_1 as the actual parameter for CI_StackItemType, math type STRING would refer to a string of math type INTEGER. In this instance, the mathematical model of Stack would be a string of INTEGER values.

[11] The stack specified in Figure 3.10 is a *homogeneous* stack. That is, all elements in instances of the specified stack are objects of a single fixed type.

```
specification AT_Stack

   context
      uses  MT_String_Theory
      uses  AI_AnyType
      needs CI_StackItemType implementing AI_AnyType

   auxiliary
      math module MI_String_Theory is MT_String_Theory
                  with (StringItemType => AnyType)

   interface

      type Stack is modeled by STRING
         exemplar  s
         initially s = EMPTY_STRING

      procedure Push (s : Stack, x : AnyType)
         ensures  s = #s * <x>  and  x = #x

      procedure Pop (s : Stack, x : AnyType)
         requires s /= EMPTY_STRING
         ensures  #s = s * <x>

      function Length (s : Stack) : Integer
         ensures Length = |s|

end AT_Stack
```

Figure 3.10: Abstract Template AT_Stack

The interface section of AT_Stack defines the program type Stack and three operations: Push, Pop, and Length. As noted above, type Stack is modeled by a mathematical string of elements. The initial value of a Stack object is modeled by an empty string. The (constant) math operation EMPTY_STRING is provided by MI_String_Theory. The post-condition of Push specifies that after the Push operation completes, the new value of the stack (s) is modeled by the old value of the stack (#s) concatenated with (*) the singleton string containing the element pushed onto the stack (<x>). Also, the value modeling the element being pushed, is the same before and after the operation (x = #x)[12]. Pop's requires clause specifies the pre-condition that the stack not be empty. Pop's ensures clause specifies that after the

[12]The post-condition of Pop in a RESOLVE-style stack interface does not require that the element being pushed is unchanged. We discuss the RESOLVE approach in the context of a queue component in Section 5.2.

Pop operation, the new value of the stack (s) concatenated with the singleton string containing the element popped from the stack (<x>) will be the same as the old value modeling the stack. The Length operation is a function which returns the length of the stack, modeled by the length of the string representing the stack (|s|).

Figure 3.11 shows the encoding of concrete template CT_Stack_1 which provides an implementation of the behavior specified by AT_Stack. CT_Stack_1 has two deferred dependencies. First, it needs an implementation of AI_AnyType to provide the type of elements to be contained in the stack. Second, it needs an implementation of *an instance of* AT_One_Way_List that has been instantiated with the same concrete instance supplied as the first parameter, the actual parameter corresponding to CI_StackItemType. The list implementation supplied as the second parameter is used as the stack representation. AT_One_Way_List specifies a generic list which is modeled by a pair of STRING values, a left string and a right string, both initially empty. The insertion and removal point for elements in the list is the leftmost element of the right string. A more detailed description of the One_Way_List specification may be found in [SWH93].

The operations Push, Pop, and Length are implemented trivially by calling the list operations Add_Right, Remove_Right, and Right_Length respectively. This implementation only uses the right string of the list which grows to the *left* as elements are added. Since Stack is modeled by a string that grows to the *right* as elements are pushed onto it, the correspondence clause states that the right string modeling the stack's list representation, when reversed, corresponds to the string modeling the stack. Note that the math operation REVERSE is defined in MT_String_Theory and is available for use here only because CT_Stack_1 **uses** AT_Stack which **uses** MT_String_Theory.

Figure 3.12 is a CCD showing the behavioral relationships between CT_Stack_1, AT_Stack and the other components upon which it directly depends. No **uses** relationships are shown since these are implied by the **needs** relationships. Also, we do not show dependencies on components providing built-in types such as CI_Integer_1 which defines the type Integer returned by the Length function.

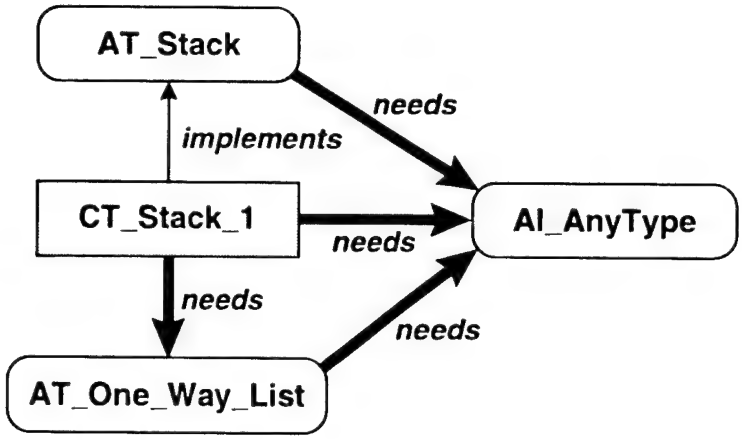
We claim that CT_Stack_1 **implements** AT_Stack by the third definition of **implements** (on page 46). That is, for any instantiation C of CT_Stack_1 there exists *some* instantiation T of AT_Stack, such that **imps**(C, T) holds. In this case, if the same concrete instance is supplied as the actual parameter corresponding to CI_StackItemType for both AT_Stack and CT_Stack_1, then the concrete instance described by the instantiation of CT_Stack_1 will **implement** the abstract instance described by the instantiation of AT_Stack.

Figure 3.13 shows the code for two instantiations used to generate a stack of flip-flops. The first instantiation describes the concrete instance CI_Flipflop_List

```
implementation CT_Stack_1
   implements  AT_Stack

   context
      uses  AI_AnyType
      uses  AT_One_Way_List
      needs CI_StackItemType implementing AI_AnyType
      needs CI_List implementing AT_One_Way_List with
               (CI_ListItemType => CI_StackItemType)

   interface

      type Stack is represented by
            holder : List
         end representation
         exemplar s_rep
         correspondence s = REVERSE(s_rep.holder.right)

      procedure Push (s : Stack, x : AnyType)
         begin
            Add_Right(s.holder)
         end Push

      procedure Pop (s : Stack, x : AnyType)
         begin
            Remove_Right(s.holder)
         end Pop

      function Length (s : Stack) : Integer
         begin
            return (Right_Length(s.holder))
         end Length

end CT_Stack_1
```

Figure 3.11: Concrete Template CT_Stack_1

which implements the behavior specified by the abstract instance generated by instantiating AT_One_Way_List with concrete instance CI_Flipflop_3 (shown in Figure 3.5). The implementation of CI_Flipflop_List is provided by the instantiation of CT_One_Way_List_1 with CI_Flipflop_3. The concrete instance CI_Flipflop_Stack is generated similarly. It implements the abstract instance formed by instantiating AT_Stack with CI_Flipflop_3. The implementation of CI_Flipflop_Stack is provided by instantiating CT_Stack_1 with CI_Flipflop_3 serving as the element type and CI_Flipflop_List serving as the list implementation used to represent the stack.




```
implementation CI_Flipflop_List
   implements AT_One_Way_List with
      (CI_ListItemType => CI_Flipflop_3)
   by CT_One_Way_List_1 with
      (CI_ListItemType => CI_Flipflop_3)

implementation CI_Flipflop_Stack
   implements AT_Stack with
      (CI_StackItemType => CI_Flipflop_3)
   by CT_Stack_1 with
      (CI_StackItemType => CI_Flipflop_3,
                CI_List => CI_Flipflop_List)
```

Figure 3.13: Instantiation of CT_Stack_1

The **needs** relationship between an implementation and a specification expresses a *polymorphic* relationship. Many different "forms" or implementations of the abstract component operations may be used by the objects declared from instances of the concrete component. There are two primary strategies for encoding this polymorphism in programming languages. Object-oriented disciplines typically use *inheritance* and *dynamic binding* to achieve polymorphism. The notation used in this chapter uses parameterization and static binding. RESOLVE/Ada95, discussed in Chapter 5 uses a combination of parameterization and inheritance. Section 4.2 discusses programming language support for expressing the **needs** relationship. In the next section

we discuss the benefits of designing components using the **implements** and **needs** relationships.

3.4.3 Integration Dependencies Versus Design Dependencies

Recall from Chapter 1 the difference between *design dependencies* and *integration dependencies*. A component's design dependencies exist independent of any particular use of the component. A component's integration dependencies are the dependencies it has on other components once integrated into a component-based system. The **uses** relationship between concrete instances naturally expresses integration dependencies and also may express fixed design dependencies. For example, the relationship `CI_Threeway_1` **uses** `CI_Flipflop_3`, as encoded in Figure 3.7, expresses a fixed design dependency. The **needs** relationship may be used to express design dependencies as deferred dependencies. The relationship `CT_Threeway_1` **needs** `AI_Flipflop`, as encoded in Figure 3.8 and depicted in Figure 3.9, expresses a deferred design dependency. Building a system from concrete templates with deferred dependencies may require more effort than building a system from (pre-existing) concrete instances with only fixed dependencies, since templates must be instantiated. However, as we discuss below, using **needs** relationships instead of **uses** relationships to express design dependencies can significantly improve the understandability, maintainability, and reusability of components.

Figure 3.14 shows three different views of the same component-based system, each emphasizing a different component relationship. Figure 3.14(a) is the same diagram shown in Figure 2.8 and explained in Section 2.4.1. The arrows represent the (direct) **uses** relationships involving these five concrete instances. The arrows shown in Figure 3.14(b) represent the **needs** relationship. This diagram shows that concrete template `A1T` needs a component that **implements** the behavioral specification `B` and one that **implements** the behavioral specification `C`. In the system shown, instantiations of `B1T` and `C1T` satisfy these requirements, respectively. `B1T`'s requirement for implementations of `D` and `E` and `C1T`'s requirement for an implementation of `E` are satisfied by implementations `D1`, `E1`, and `E1`, respectively. The arrows in Figure 3.14(c) represent the **implements** relationship and also the template instantiations necessary to build the system. This view conveys exactly the same information as Figure 3.14(b), but more clearly depicts `A1T`, `B1T`, and `C1T` as templates with "holes" to be filled in.

Figure 3.14(a) is a traditional *structure chart* that shows which parts of the system depend on which other parts. It does not, however, provide any information about the role played by each component or what the options might be for component replacement. Figure 3.14(b) and (c) convey more information useful to system maintainers and component composition tools. These diagrams provide information about the roles played by each sub-component of the system (although no system-level specification describing the behavior of `A1T` is shown). By depicting external

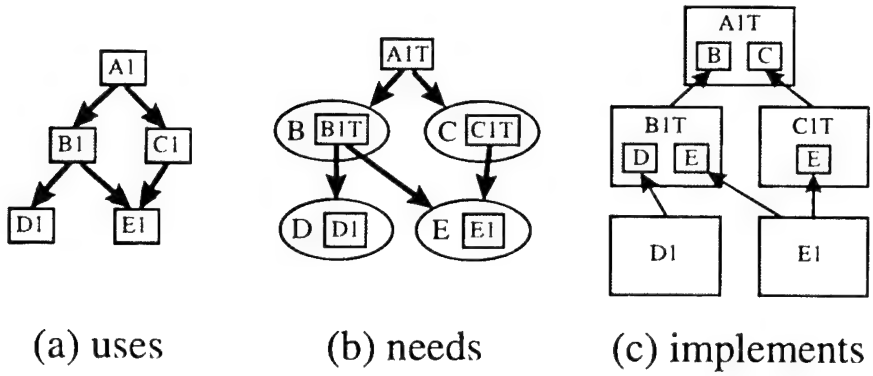


Figure 3.14: Three Views Of The Same System

dependencies of A1T, B1T, and C1T at the specification level. Figures 3.14(b) and (c) identify requirements for components that might replace B1T, C1T, D1, or E1.

Consider an example of how the system in Figure 3.14 might be modified. Assume that the system performance needs to be improved and that analysis determines that component E1 is a bottleneck. The figure shows that B1T and C1T both depend on *any* concrete instance that **implements** E, not specifically on implementation E1. Thus, another component that **implements** E, say E2, can be substituted for E1 in this system without requiring changes to any of the other components. Presumably replacing E1 with E2 would yield better system performance due to differences between the two implementations not constrained by E.

As another example, implementation D1 might serve as an interface to the system's environment. That is, D1 might interact directly with operating system software or hardware. If the system needs to be rehosted to a new environment, then component D1 might need to be replaced with another implementation providing the same behavior as specified by D, but implementing these services differently in order to interact with a different environment. In this case, another component, say D2, that interacts with the new environment and which *implements* D could be substituted for D1 without requiring changes to other components in the system.

We said that each of the three diagrams in Figure 3.14 depicts the same *specific* system composed of five components. For this to be the case, B1 must denote the component generated by instantiating B1T with D1 and E1, C1 must denote the component generated by instantiating C1T with E1, and A1 must denote the component (or top-level program unit) generated by instantiating A1T with B1 and C1. In this case, none of the five integration dependencies shown in Figure 3.14(a) need be design dependencies. That is, prior to instantiation of B1T, there may be no component B1

that specifically depends on D1 and E1. If the system were developed from components in a component library, then the library might contain the components A1T, B1T, C1T, D1, and E1 with design dependencies expressed in terms of **implements** and **needs** rather than in terms of **uses**.

Software engineers generally want to avoid any unnecessary dependencies (coupling) between components. The **needs** relationship between an implementation and a specification may be applied to achieve this goal. Figure 3.9 depicts the **needs** relationship between the implementation CT_Threeway_1 and the specification AI_Flipflop. In this case, the description of the three-way switch implementation only depends on the abstract description of a flip-flop (provided by AI_Flipflop). During program execution, an object declared from an instance of CT_Threeway_1 will indeed depend on some specific implementation of AI_Flipflop.

Fully understanding the non-functional characteristics of this object would include understanding the non-functional characteristics of the specific flip-flop implementation used. However, to understand and reason about the functional behavior described by an instance of CT_Threeway_1, it is sufficient to understand the behavior described by AI_Flipflop and AI_Threeway (assuming CT_Threeway_1 **implements** AI_Threeway as claimed).

3.5 The extends Relationship

Once a software system has been designed and implemented, changes to requirements are likely to call for improved performance and functionality. Improvements in performance which do not alter functionality may be addressed by replacing one concrete component with another. This section describes the **extends** relationship which is useful for adding new functionality to existing components.

If component designers had perfect foresight they might be tempted to design components with all the functionality that clients could ever want. If this were possible, then **implements**, **needs**, and **uses** might be sufficient for describing component relationships. Unfortunately, component designers do not have perfect foresight. No matter how much forethought designers apply, new requirements almost always expose some desirable and unforeseen functionality. Clearly we need some method for extending the functionality of components already in use.

Lack of perfect foresight, however, is not the only reason for providing a means to extend the functionality of components. Some disciplines for designing components advocate providing a wide assortment of possibly useful operations in each individual component [Mey94]. The problem with this approach is that the resulting interfaces are more complex and thus more difficult for clients and component implementers to understand. Furthermore, this approach either leads to code bloat resulting from many unused operations or requires assumptions about optimizations which attempt to expunge the code of unused operations. One reasonable approach is to design

components with a minimally sufficient set of operations which lay a foundation upon which future enhancements may be constructed. Whatever initial approach is used, however, enhancements to functionality are inevitable.

3.5.1 Extension Components

The **extends** relationship is a behavioral relationship between two abstract components. The **extends** relationship may be defined informally as follows:

> Abstract component A_2 **extends** abstract component A_1 if and only if every concrete component which **implements** A_2 also **implements** A_1.

The **extends** relationship is a conformance relationship modeled by the **exts** relation defined in Equation 2.3. However, like **imps**, **exts** only describes a relation over instance components. As we did with **implements**, we overload the term **extends** to include relationships involving template components. We use the overloaded term "`extends`" for the following three distinct relationships.

- If A_1 and A_2 are both abstract instances, then the claim that A_2 **extends** A_1 is an assertion that **exts**(A_2, A_1) holds.
- If A_1 is an abstract instance and A_2 is an abstract template, then the claim that A_2 **extends** A_1 is an assertion that for *any* instantiation A_2' of A_2, **exts**(A_2', A_1) holds.
- If A_1 and A_2 are both abstract templates, then the claim that A_2 **extends** A_1 is an assertion that for *any* instantiation A_2' of A_2 there exists *some* instantiation A_1' of A_1, such that **exts**(A_2', A_1') holds.

We discuss an example of the first case in this section. In Section 3.5.3, we discuss an example of the third case. The second case is included primarily for completeness. We are not aware of any motivating examples for this form of **extends**.

In Section 2.3.2 we discussed three different ways in which a specification component might be extended: *specialization*, *generalization*, and *augmentation*. If specification components are parameterized, then specialization of specifications may be achieved conveniently through the instantiation of abstract templates. If specification components are well-designed from a component-based reuse perspective, then there should be little need for generalization (weakening operation pre-conditions). Therefore, in this section we focus on extension by augmentation, that is, extending a component by adding new operations.

As a simple example of the **extends** relationship, consider the abstract instance `AI_Flipflop` shown in Figure 3.1 and the abstract instance `AI_FFExt` shown in Figure 3.15. We claim that `AI_FFExt` **extends** `AI_Flipflop` in accordance with the first

```
specification AI_FFExt
   extends AI_Flipflop

   interface

      type Flipflop is modeled by BOOLEAN
         exemplar  ff
         initially ff = FALSE

      procedure Toggle (f : Flipflop)
         ensures f = NOT #f

      function Test (f : Flipflop) : Boolean
         ensures  Test = f

      procedure Set (f : Flipflop)
         ensures f = TRUE

end AI_FFExt
```

Figure 3.15: Abstract Instance AI_FFExt

of the three definitions of **extends** above. That is, any concrete instance that **implements** AI_FFExt also **implements** abstract instance AI_Flipflop. In this case, as is often the case with extension by augmentation, it is easy to see that the **extends** relationship is justified. The only difference between AI_Flipflop and AI_FFExt (aside from their names) is that AI_FFExt includes the extends clause in its header and a specification of the Set operation. The extends clause plays the same role as the implements clause discussed in Section 3.3: it records design intent and it identifies syntactic and semantic conformance checking requirements. The specification of procedure Set states that the abstract value of the flipflop passed as an argument is TRUE after execution of the operation. Thus AI_FFExt provides the same type and operations as AI_Flipflop plus the additional operation Set.

From a software maintenance perspective, the relationship between AI_FFExt and AI_Flipflop raises an important issue. Notice that AI_FFExt includes a *copy* of the interface section of AI_Flipflop. This method of encoding a specification extension, duplicating the specification being extended, has advantages and disadvantages with respect to the alternative approach of *specification-by-difference*. Figure 3.16 shows the abstract instance AI_FFWSet. AI_FFWSet specifies the same behavior as AI_FFExt using the specification-by-difference approach. AI_FFWSet **uses** AI_Flipflop and defines its own interface in terms of the interface defined by AI_Flipflop. The statement "re-exports AI_Flipflop" in Figure 3.16 includes all of the interface section of

```
specification AI_FFWSet
   extends AI_Flipflop

   context
      uses AI_Flipflop

   interface

      re-exports AI_Flipflop

      procedure Set (f : Flipflop)
         ensures f = TRUE

end AI_FFW_Set
```

Figure 3.16: Abstract Instance AI_FFWSet

AI_Flipflop in the interface section of AI_FFWSet. As we discuss in Chapter 4, re-exports is a language mechanism similar to a static form of inheritance.

The primary difference between AI_FFExt and AI_FFWSet is that the behavior described by AI_FFWSet depends upon AI_Flipflop whereas the behavior described by AI_FFExt does not depend on AI_Flipflop. Note that the extends clause in both of these components is only a *claim* documenting an intended relationship and does not influence the behavior described by either of the components. An advantage of including the text of the extended specification's interface in the component extending the specification is that the extended specification is more cohesive. A software engineer attempting to understand the behavior specified by AI_FFExt need only look at that single component[13]. In order to understand the behavior described by AI_FFWSet, both AI_FFWSet and AI_Flipflop must be examined. In the case where one specification **extends** more than one other specification, it may be necessary to examine several specification components in order to fully understand the behavior described by a single extension component.

An advantage of the specification-by-difference approach is that a specification extension component is simpler and focuses attention on the specification of the added behavior. Like implementation units, large complex specifications may be easier to understand when divided up into smaller units. Also, with the specification-by-difference approach, there may be only a single point of modification if a specification component needs to be changed. If the interface of a specification component with many

[13]This assumes, of course, that the maintainer understands the behavior of the built-in operations for Integer and Boolean.

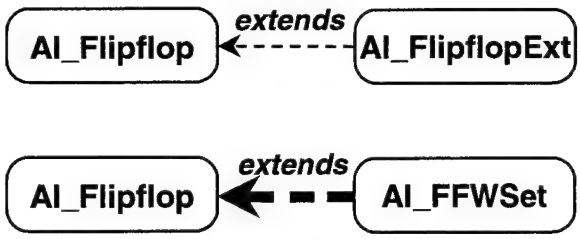


Figure 3.17: The **extends** Relationship Without and With Coupling

extensions needs to be changed, then all copies of that interface embedded within extension specification components probably need to be changed as well. Of course, any change to a specification component potentially requires re-justification of `extends` relationships between that component and others.

One way to mitigate the problem of having a single interface composed of operations defined in various components is to use a browser tool capable of displaying a complete interface even if its constituent parts are defined in several different components. Such a tool could use the information recorded by `re-exports` clauses in order to determine the full extent of an extended interface. Meyer describes a tool with similar capabilities for viewing Eiffel components extended by inheritance in the "flat-short" form [Mey94, p. 29]. In the examples which follow, we use the specification-by-difference approach to specification extension. It is important to realize, however, that the **extends** relationship is independent of the language mechanisms, such as `re-exports` or inheritance, used to encode a specification extension. As we discuss in Chapter 4, programming languages generally do not make this distinction between behavioral and syntactic relationships.

The top of Figure 3.17 shows the CCD depiction of the `extends` relationship between `AI_FFExt` and `AI_Flipflop`. The **extends** relationship is depicted as a *thin* dashed arrow from the extension specification to the specification being extended. In addition to its **extends** relationship with `AI_Flipflop`, `AI_FFWSet` also **uses** `AI_Flipflop`. Since specification-by-difference is the most common way of extending a specification, rather than drawing two arrows, we depict the combined **extends** and **uses** relationships with a *thick* dashed arrow as shown in the bottom of Figure 3.17. The thick arrow indicates that `AI_FFWSet` depends on (is coupled to) `AI_Flipflop`.

3.5.2 Implementing Extension Components

We now discuss three different approaches to encoding an implementation of an abstract extension component: the *direct*, *coupled*, and *layered* approaches. The primary motivation here is to demonstrate another use of the **needs** relationship layered extensions. The discussion also provides more examples of the `implements` and `uses` relationships and raises some interesting design and implementation issues.

Figure 3.18 shows `CI_FFWSet_1`, a concrete instance that **implements** `AI_FFWSet`. `CI_FFWSet_1` is a *direct* implementation of `AI_FFWSet`. A direct implementation of a specification does not depend on any other implementations of the specification being extended. Thus a direct implementation must implement the union of the behavior specified by all extension components that it implements including that of the behavior being extended. In this example, `CI_FFWSet_1` must provide a representation for the type `Flipflop` and implementations for the operations `Toggle` and `Test` as well as an implementation for the extension operation `Set`. The implementation shown in Figure 3.18 uses the obvious representation, a `Boolean` program type represents the `BOOLEAN` math type used to model a flipflop. We will discuss the advantages and disadvantages of the direct implementation approach after looking at examples of the other two approaches.

Figure 3.19 shows `CI_FFWSet_2`, another component that **implements** `AI_FFWSet`. `CI_FFWSet_2` is a *coupled* implementation of `AI_FFWSet`. A coupled implementation of an extension has a fixed dependency on a specific implementation of the specification being extended. That is, the extension implementation **uses** an implementation of the specification being extended. In this case, `CI_FFWSet_2` has a fixed dependency on `CI_Flipflop_2` (Figure 3.2) which **implements** `AI_Flipflop`.

The primary motivation for a coupled implementation of an extension is to allow the extension implementation to have direct access to the data representation of a specific implementation of the component being extended. Thus, as with direct implementations, new operations may be implemented in terms of a specific data representation. For example, the `Set` operation defined in `CI_FFWSet_2` uses the `Integer` representation of type `Flipflop` defined in `CI_Flipflop_2`. As we discuss in Chapter 4, different programming languages support a variety of different mechanisms that allow or disallow one component to have direct access to a type representation defined in another component. A coupled implementation of an extension is only possible when the implementation being extended is encoded in a way that allows another component to directly access the data representation of a type that it defines.

`CI_FFWSet_2` has direct access to `CI_Flipflop_2`'s representation of `Flipflop` because `CI_FFWSet_2` **uses** `CI_Flipflop_2` and `CI_Flipflop_2` defines its representation of `Flipflop` in the `interface` section (as opposed to the `auxiliary` section). The "`re-exports CI_Flipflop_2`" statement includes the `interface` section of `CI_Flipflop_2` in the `interface` section of `CI_FFWSet_2`. This use of `re-exports`

```
implementation CI_FFWSet_1
   implements  AI_FFWSet

   interface

      type Flipflop is represented by
            state : Boolean := False
         end representation
         exemplar ff_rep
         correspondence ff = ff_rep.state

      procedure Toggle (f : Flipflop) is
         begin
            f.state := not(f.state)
         end Toggle

      function Test (f : Flipflop) : Boolean is
         begin
            return f.state
         end Test

      procedure Set (f : Flipflop) is
         begin
            f.state := true
         end Set

end CI_FFWSet_1
```

Figure 3.18: `CI_FFWSet_1` — A Direct Implementation

at the *implementation level* is analogous to the use of `re-exports` by `AI_FFWSet` (Figure 3.16) at the specification level. Whereas re-exporting an abstract component may be used for achieving specification-by-difference, re-exporting a concrete component is useful for achieving implementation-by-difference. Implementation-by-difference is one way to support reuse of existing implementation code and is a primary use of inheritance.

Figure 3.20 shows `CT_FFWSet_3`, a concrete *template* that **implements** `AI_FFWSet`. That is, every concrete instance described by instantiating `CT_FFWSet_3` **implements** `AI_FFWSet`. `CT_FFWSet_3` is a *layered* implementation of `AI_FFWSet`. A layered implementation of an extension has a deferred dependency on an implementation of the specification being extended. That is, the extension implementation **needs** an implementation of the specification being extended. In this example, `CT_FFWSet_3` has a deferred dependency on an implementation of `AI_Flipflop`.

```
implementation CI_FFWSet_2
   implements  AI_FFWSet

   context
      uses CI_Flipflop_2

   interface

      re-exports CI_Flipflop_2

      procedure Set (f : Flipflop) is
         begin
             f.state := 1
         end Set

end CI_FFWSet_2
```

Figure 3.19: CI_FFWSet_2 A Coupled Implementation

Like a coupled implementation of an extension, an *instantiation* of a layered implementation reuses a specific implementation of the specification being extended. However, since a layered implementation may be instantiated with *any* concrete instance that **implements** the specification being extended, it only has access to the interface defined by the specification it **needs**. Thus a layered implementation does not have direct access to the concrete representation of any types defined in the component being extended. Operations defined in a layered implementation of an extension must be encoded in terms of "layered" on top of operations provided by the component being extended. Note that we have already seen examples of layering used to implement one abstraction in terms of another: CT_Threeway_1 (Figure 3.8) and CT_Stack_1 (Figure 3.11).

For CT_FFWSet_3, shown in Figure 3.20, the **interface** section must be defined in terms of the abstract interface provided by AI_Flipflop. Thus the operation Set is implemented in terms of the operations Toggle and Test, as specified by AI_Flipflop, rather than in terms of a specific data representation. The "re-exports CI_Flipflop" statement includes the **interface** section of the concrete instance used to instantiate CT_FFWSet_3 as part of the interface described by an *instantiation of* CT_FFWSet_3. Thus the concrete instance described by an instantiation of CT_FFWSet_3 provides everything in the interface of the concrete instance which serves at the actual parameter for CI_Flipflop, plus an implementation of the operation Set. Since a concrete instance that **implements** AI_Flipflop may implement more behavior than that specified by AI_Flipflop, an instantiation of CT_FFWSet_3 may

```
implementation CT_FFWSet_3
   implements  AI_FFWSet

   context
      uses  AI_Flipflop
      needs CI_Flipflop implementing AI_Flipflop

   interface

      re-exports CI_Flipflop

      procedure Set (f : Flipflop) is
         begin
            if not(Test(f)) then
               Toggle(f)
            end if
         end Set

end CT_FFWSet_3
```

Figure 3.20: CT_FFWSet_3 — A Layered Implementation

correspondingly implement more behavior than that specified by AI_FFWSet. Regardless of how CT_FFWSet_3 is instantiated, however, its contents may only refer to types and operations as specified in AI_Flipflop (and, of course, built-in types and operations).

Figure 3.21 shows the code for three instantiations demonstrating how a layered implementation of an extension may be instantiated and the resulting concrete instance further extended. The first instantiation defines the concrete instance CI_FFBase which describes the behavior specified by AI_Flipflop (the type Flipflop and operations Toggle and Test) as implemented by CI_Flipflop_2. The second instantiation defines the concrete instance CI_FFWSet which describes the behavior specified by AI_FFWSet as implemented by CT_FFWSet_3 instantiated with CI_FFBase. Thus, CI_FFWSet provides implementations of Toggle and Test as described by CI_Flipflop_2 and an implementation of Set realized by calls to those implementations of Toggle and Test.

The third instantiation defines the concrete instance CI_FFWSetReset using the specification AI_FFWReset and the implementation CT_FFWReset_3. AI_FFWReset **extends** AI_Flipflop with the operation Reset in the same way as AI_FFWSet does with Set. CT_FFWReset_3 **implements** AI_FFWReset in the same manner CI_FFWSet_1 **implements** AI_FFWSet. (The code for AI_FFWReset and CT_FFWReset_3 is not shown.) CI_FFWSetReset describes the behavior specified by the union of AI_FFWSet and

```
implementation CI_FFBase
   implements AI_Flipflop
   by CI_Flipflop_2

implementation CI_FFWSet
   implements AI_FFWSet
   by CT_FFWSet_3 with
      (CI_Flipflop => CI_FFBase)

implementation CI_FFWSetReset
   implements AI_FFWSet, AI_FFWReset
   by CT_FFWReset_3 with
      (CI_Flipflop => CI_FFWSet)
```

Figure 3.21: Instantiation of Layered Extension Implementations

AI_FFWReset as implemented by CT_FFWReset_3 instantiated with CI_FFWSet, the result of the second instantiation. Thus, CI_FFWSetReset provides the concrete type Flipflop and operations Toggle, Test, Set, and Reset.

Note that the direct and coupled implementations of AI_FFWSet could be instantiated in much the same way as the first instantiation shown in Figure 3.21. In both cases "implements" would be followed with "AI_FFWSet". For the direct implementation, "by" would be followed by "CI_FFWSet_1". For the coupled implementation, "by" would be followed by "CI_FFWSet_2".

Figure 3.22 is a CCD depicting the three different implementations of AI_FFWSet as well two implementations of AI_Flipflop and the relationships among these components. CI_FFWSet_1 has no dependencies on other components (except for the built-in component CI_Boolean_1 which we do not show). Thus modifications to other components cannot affect the behavior described by CI_FFWSet_1. If AI_FFWSet or AI_Flipflop were modified, then the correctness of the claim that CI_FFWSet_1 **implements** AI_FFWSet (and implicitly AI_Flipflop) might change, but not the behavior implemented by CI_FFWSet_1. The **uses** relationship shown between CI_FFWSet_2 and CI_Flipflop_2 depicts the fixed dependency of CI_FFWSet_2 on CI_Flipflop_2. This indicates that a modification to CI_Flipflop_2 may alter the behavior described by CI_FFWSet_2. The **needs** relationship shown between CT_FFWSet_3 and AI_Flipflop depicts the deferred dependency of CT_FFWSet_3 on AI_Flipflop. That is, CT_FFWSet_3 needs an implementation of AI_Flipflop. A change to the specification AI_Flipflop could affect the behavior described by CT_Flipflop_3. However, changes to any implementations of AI_Flipflop cannot alter the behavior described by CT_FFWSet_3.

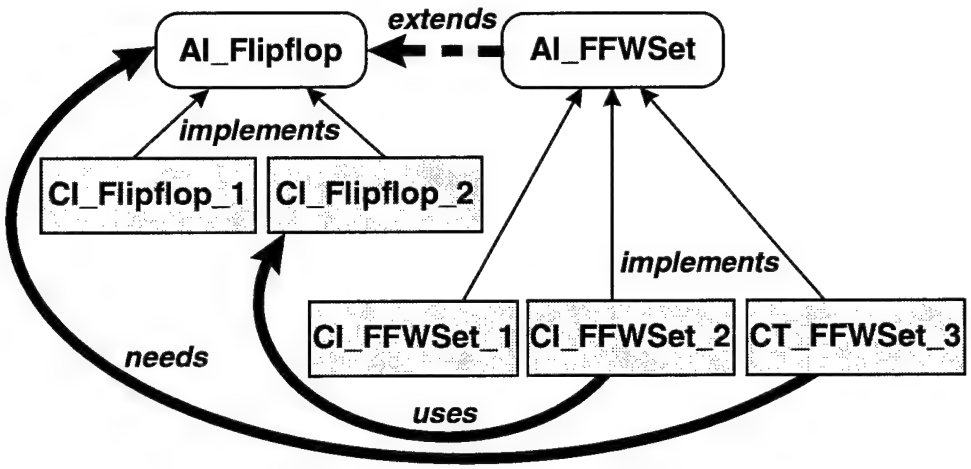


Figure 3.22: Three Ways To Implement An Extension

We now address the relative advantages and disadvantages of each of the three approaches to implementing an extension. The direct approach, exemplified by `CI_FFWSet_1` in Figure 3.18, is generally the least attractive of the three alternatives since it requires implementing the component being extended, in addition to the extension, from scratch. Both the coupled and layered approaches have the advantage of reusing existing implementation code. The principal advantage of the direct approach is that the code implementing the extension operations can have direct access to data representations which cannot be accessed directly using the layered approach and which may not be accessible at all to other components (in which case the coupled approach cannot be used). With direct access to the representation, it may be possible to implement some extension operations much more efficiently than is possible with layering. While there is unlikely to be a performance advantage using a direct implementation in the case of `CI_FFWSet_1`, we present an example in Section 3.5.3 where the direct approach does provide a significant performance improvement over the layered approach.

The coupled approach, exemplified by `CI_FFWSet_2` in Figure 3.19, offers the principal advantage of the direct approach, direct access of extension operations to data representations, and one of the main advantages of the layered approach, reuse of existing implementation code. Thus in some cases, a coupled implementation may be significantly more efficient than a layered alternative and less costly to develop than a direct implementation. A disadvantage of the coupled approach is that it typically requires "privileged" access to an existing implementation component in order to break encapsulation and gain direct access to data representations of types defined within

the implementation being extended. We discuss this issue more in Chapter 4. In a software components industry in which implementation components are distributed in "black box" object-code format, coupled implementations of extensions are not an option except for organizations which have the source code of the implementation being extended. Finally, justifying that a coupled implementation correctly **implements** an extension specification may require significant effort which is not required when the layered approach is used. Edwards discusses the subtleties of this issue in terms of what he calls "representation inheritance", a form of extension by coupled implementation [Edw96].

The layered approach, exemplified by `CT_FFWSet_3` in Figure 3.20, offers significant advantages over the direct and coupled approaches with relatively minor drawbacks. As with the coupled approach, the layered approach reuses the code of an existing implementation of the specification being extended. However, the layered approach does not require privileged access to the implementation being extended since it views the implementation in terms of an abstract interface. A major benefit of the layered extension implementation is that it can be used to extend *any* implementation of the specification being extended, not just a single implementation as is the case with the coupled approach. This makes it possible to "chain" together multiple extensions as shown in Figure 3.21. Layering also insulates an extension from modifications to implementations of the specification being extended.

The extension implementation task should be conceptually easier with the layered approach since a software engineer implementing layered extension operations only needs to understand the behavioral specification of the implementation being extended and not any of its implementation details, such as data representations. An empirical study by Zweben, et. al., provides evidence that the layering approach can lead to higher programmer productivity and lower defect rates [ZEWH95]. As mentioned above, justifying that an implementation extension is correct with respect to the specification it implements is typically easier for layered implementations than for coupled implementations. The reason for this is that operations implemented by layering cannot violate representation invariants of the implementation being extended as long as the underlying operations are correctly implemented and called only when their preconditions are satisfied.

As noted above, the primary disadvantage of layered implementations is that they may be less efficient than comparable direct or coupled implementations. The inefficiency arises from the additional operation calling overhead required to invoke operations of the implementation being extended and, in some cases, arises from a significantly increased algorithmic complexity resulting from the limitations of indirect data manipulation. For example, the implementations of `Set` described by `CI_FFWSet_1` (Figure 3.18) and `CI_FFWSet_2` (Figure 3.19) are likely to be more efficient than the layered implementation of `Set` described by `CI_FFWSet` (Figure 3.21).

but only by a constant factor. In some cases, the additional operation calling overhead resulting from layering may be reduced by using optimization techniques such as in-lining of operations. Another disadvantage inherent to the layering approach is that clients of layered extensions must instantiate the concrete template defining a layered implementation in order to generate a useful concrete instance. This problem may be mitigated by "pre-instantiating" components using a technique called *partial instantiation*. We discuss partial instantiation in Chapter 5.

The layered approach to implementing extension specifications is the primary approach used by the RESOLVE discipline [SW94, p. 41]. In Section 5.6 we present examples of how layered extensions are encoded in RESOLVE/Ada95.

3.5.3 Extension Of Template Components

On page 64 we listed three distinct relationships which apply to the overloaded term **extends**. The **extends** relationship between two abstract instances was exemplified by the relationship `AI_FFWSet` **extends** `AI_Flipflop`. The second relationship, an abstract template that **extends** an abstract instance, could be used to express a deferred dependency on a component used by an extension, but not by the abstract instance that the extension **extends**. We include this second definition of **extends** for completeness and do not provide an example. In this section we discuss the third **extends** relationship between two abstract templates.

Figure 3.23 shows the abstract template `AT_SWRev` that **extends** `AT_Stack` (shown in Figure 3.10) in accordance with the third, template-to-template, definition of **extends**. Since `AT_Stack` is an abstract template, `AT_SWRev` must also be an abstract template. This is the only way that all of the interface defined by `AT_Stack` can be included in the interface defined by `AT_SWRev` as necessary for justifying the `extends` relationship. Like `AT_Stack`, `AT_SWRev` **needs** a concrete instance that `implements` `AI_AnyType` to provide a definition of the type of element contained within the stack. The type corresponding to `AnyType` in the actual parameter bound to `CI_StackWRItemType`[14] determines the type of elements contained in the stack. `AT_SWRev` **uses** `AT_Stack` in order to define its interface in terms of the interface specified by `AT_Stack`. However, since `AT_Stack` is an abstract *template*, its interface section cannot be re-exported directly as was the case with `AI_FFWSet` re-exporting the interface of `AI_Flipflop` (Figure 3.16).

Since `AT_Stack` is a template, the meaning of its `interface` section depends on the binding of its formal parameter `CI_StackItemType`. Thus, in order for the re-exported interface of `AT_Stack` to have any meaning, `CI_StackItemType` must be bound to some concrete instance. Furthermore, for an instantiation of `AT_SWRev`

[14]Note that any formal parameter name would do where we use `CI_StackWRItemType`. Normally, we would use the *same* formal parameter name here as in `AT_Stack`. However, in this example we use different formal parameter names in order to make the code easier to understand.

```
specification AT_SWRev
   extends AT_Stack

   context
      uses  AT_Stack
      uses  AI_AnyType
      needs CI_StackWRItemType implementing AI_AnyType

   interface

      re-exports AT_Stack with
         (CI_StackItemType => CI_StackWRItemType)

      procedure Reverse (s : Stack)
         ensures  s = REVERSE(#s)

end AT_SWRev
```

Figure 3.23: An Extension of An Abstract Template

to **extend** an instantiation of AT_Stack, both specifications clearly need to specify stacks with the same type of elements. The re-exports clause in AT_SWRev addresses both of these issues by describing the re-exported interface as AT_Stack instantiated with CI_StackWRItemType serving as the actual parameter for CI_StackItemType. That is, the interface described by an instance of AT_SWRev includes the interface described by an instance of AT_Stack instantiated with the same concrete instance used to instantiate AT_SWRev.

Recall that the definition of **extends** between two abstract templates requires that for *any* instantiation of AT_Stack, say AI_1, there exists *some* instantiation of AT_SWRev, say AI_2, such that AI_2 **extends** AI_1 (by the definition of **extends** between two abstract instances). We claim that AT_SWRev **extends** AT_Stack for the following reason. Assume that AT_Stack is instantiated with the concrete instance CI serving as the actual parameter corresponding to AT_Stack's formal parameter CI_StackItemType. Then any instantiation of AT_SWRev with CI as the actual parameter corresponding to AT_SWRev's formal parameter CI_StackWRItemType **extends** the instantiation of AT_Stack with CI (by the definition of **extends** between two abstract instances). That is, if AT_Stack and AT_SWRev are instantiated with the same concrete instance, then the instantiation of AT_SWRev **extends** the instantiation of AT_Stack. This implies that any concrete instance that **implements** an instantiation of AT_SWRev also **implements** the corresponding instantiation of AT_Stack. We will look at an example of how these components may be instantiated after looking at a component that **implements** AT_SWRev.

```
implementation CT_SWRev_1
   implements  AT_SWRev

   context
      uses  AI_AnyType
      uses  AT_Stack
      uses  AT_Queue
      needs CI_StackWRItemType implementing AI_AnyType
      needs CI_Stack implementing AT_Stack with
               (CI_StackItemType => CI_StackWRItemType)
      needs CI_Queue implementing AT_Queue with
               (CI_QueueItemType => CI_StackWRItemType)

   interface

      re-exports CI_Stack

      procedure Reverse (s : Stack)
              q : Queue
              x : AnyType
         begin
            for i in 1 .. Length(s) loop
               Pop(s, x)
               Enqueue(q, x)
            end loop
            for i in 1 .. Length(q) loop
               Dequeue(q, x)
               Push(s, x)
            end loop
         end Reverse

end CT_SWRev_1
```

Figure 3.24: A Layered Implementation of AT_SWRev

Figure 3.24 shows the concrete template CT_SWRev_1, which is a *layered* implementation of AT_SWRev. This implementation reverses a stack by popping each element off the stack and enqueueing it into a (first-in-first-out) queue followed by dequeueing each element from the queue and pushing it back onto the stack. CT_SWRev_1 **uses** AI_AnyType, AI_Stack, AI_Queue, and implicitly CI_Integer_1 and CI_Boolean_1. CT_SWRev_1 **needs** an implementation of AI_AnyType, and implementations of AT_Stack and AT_Queue, both instantiated with the same concrete instance serving as the implementation of AI_AnyType.

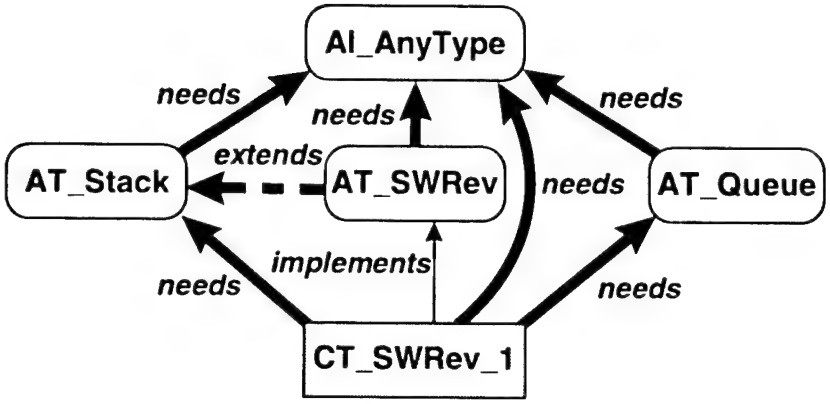


Figure 3.25: Behavioral Relationships of CT_SWRev_1

From a component maintenance perspective, we largely can ignore CT_SWRev_1's dependencies on AI_AnyType, CI_Boolean_1, and CI_Integer_1[15] since these components are fixed by the language and will not change. Thus CT_SWRev_1's **needs** relationships with AT_Stack and AT_Queue summarize the critical design dependencies for this component.

Figure 3.25 is a CCD that shows CT_SWRev_1 and the component relationships pertinent to understanding the behavior of and use of CT_SWRev_1. The thick arrows (**needs** and **extends** with **uses**) depict coupling or dependency relationships. The thin arrow (**implements**) depicts a purely behavioral relationship. The implementation behavior encoded by CT_SWRev_1 depends on the behavior specified by AT_Stack, AT_Queue, and AI_AnyType. While AT_SWRev may be used as a specification of the behavior of CT_SWRev_1, the behavior implemented by CT_SWRev_1 in no way *depends upon* AT_SWRev. Thus only changes to AT_Stack and AT_Queue could alter the behavior described by CT_SWRev_1. Note that Figure 3.25 does not encode the requirement that CT_SWRev_1, AT_Stack, and AT_Queue must all be instantiated with the *same* implementation of AI_AnyType. The component instantiation diagrams (CID's) shown in Chapter 5 convey more detailed information such as this.

Figure 3.26 shows the code describing an instantiation of CT_SWRev_1. The concrete instance CI_FFSWRev describes an implementation of a stack of flip-flops providing a stack Reverse operation. CI_FFSWRev **implements** AT_SWRev instantiated with CI_Flipflop_3. Recall that the code describing the concrete instances

[15]CT_SWRev's fixed dependency on CI_Integer_1 could be significant to a maintainer since the latter defines an implementation-dependent maximum Integer value. However, an "unbounded" stack as specified by AT_Stack would not be appropriate for applications in which the stack's length could potentially grow larger than the maximum Integer value.

```
implementation CI_FFSWRev
  implements AT_SWRev with
     (CI_StackWRItemType => CI_Flipflop_3)
  by CT_SWRev_1 with
     (CI_StackWRItemType => CI_Flipflop_3,
                CI_Stack => CI_Flipflop_Stack,
                 CI_List => CI_FLipflop_List)
```

Figure 3.26: Instantiation of CT_SWRev_1

CI_Flipflop_Stack and CI_Flipflop_List was shown in Figure 3.13. The **implements** relationship between CI_FFSWRev and AT_SWRev can be justified here only because all of the template components involved have been instantiated with the same concrete instance, CI_Flipflop_3, as the stack element type.

Since CT_SWRev_1 is a layered implementation of AT_SWRev, it may be used to extend any component that **implements** AT_Stack but does not depend on any other implementation components. Despite these advantages, the layered approach precludes an efficient constant-time Reverse operation in this case. The Reverse operation encoded in AT_SWRev executes in linear time with respect to the length of the stack being reversed. A stack Reverse operation layered on top of the interface provided by AT_Stack can do no better than linear time. A client application that needs to reverse stacks frequently, might justify creating a direct implementation of AT_SWRev with a Reverse operation that runs in constant time.

Figure 3.27 shows the concrete template CT_SWRev_2 which we claim **implements** AT_SWRev. This is a direct implementation of AT_SWRev since it defines its own stack representation and implementations for Push, Pop, and Length in addition to Reverse. CT_SWRev_2 needs an implementation of AT_Two_Way_List, a specification similar to AT_One_Way_List except that it includes a Retreat which, along with Advance, supports traversal of the list in both directions. A two-way list has the same model as a one-way list, a pair of strings. The representation of type Stack has two components: a (two-way) List labeled holder and a Boolean labeled left_top. left_top is used to keep track of which end of the list represent the top of the stack. When the value of left_top is True, the portion of the list corresponding to the right string represents the stack with the stack top being the left-most element in the right string. When the value of left_top is False, the portion of the list corresponding to the left string represents the stack with the stack top being the right-most element in the left string. Using this representation, the Reverse operation is implemented by simply changing which end of the list currently represents the top of the stack.

Achieving a constant time reverse operation does require slightly more complex and slower implementations of Push, Pop, and Length since each of these operations

```
implementation CT_SWRev_2
   implements  AT_SWRev

   context
      uses  AI_AnyType
      uses  AT_Two_Way_List
      needs CI_StackItemType implementing AI_AnyType
      needs CI_List implementing AT_Two_Way_List with
               (CI_ListItemType => CI_AnyType)
   interface
      type Stack is represented by
            holder   : List
            left_top : Boolean := True
      end representation
      exemplar s_rep
      convention if s_rep.left_top then
                     s_rep.holder.left  = EMPTY_STRING
                  else
                     s_rep.holder.right = EMPTY_STRING
      correspondence if s_rep.left_top then
                     s = REVERSE(s_rep.holder.right)
                  else
                     s = s_rep.holder.left

      procedure Push (s : Stack, x : AnyType)
         begin
            Add_Right(s.holder, x)
            if not(s.left_top) then Advance(s.holder) end if
         end Push

      procedure Pop (s : Stack, x : AnyType)
         begin
            if not(s.left_top) then Retreat(s.holder) end if
            Remove_Right(s.holder, x)
         end Pop

      function Length (s : Stack) : Integer
         begin
            if s.left_top then return (Right_Length(s.holder))
            else return (Left_Length(s.holder)) end if
         end Length

      procedure Reverse (s : Stack)
         begin
             if s.left_top then Move_To_Finish(s.holder)
             else Move_To_Start(s.holder) end if
             s.left_top := not(s.left_top)
         end Reverse
end CT_SWRev_2
```

Figure 3.27: A Direct Implementation of AT_SWRev

must test for which end of the list represents the top of the stack. An implementation of a two-way list also is likely to require more memory than a one-way list implementation. Nevertheless, if a fast reverse operation is important to the client application, then the direct implementation approach is justified in this case. Note that a coupled implementation is unlikely to be of use in this situation since the implementations `Push`, `Pop`, and `Length`, as well as the data representation, must be specifically designed for a constant-time reverse operation.

3.6 Behavioral Substitutability of Components

With the relationships we have defined, it is now very simple to characterize when one software component is substitutable for another. Since an integrated system consists of all concrete instances (e.g., Figure 3.14(a)), component-level system maintenance involves replacing one concrete instance with another. However, a maintainer cannot replace one concrete instance with another without knowing the behavioral requirements the system has for the component being replaced. Two different concrete instances may be substitutable with respect to one specification, but not with respect to another. Therefore, the substitutability relationship is a *ternary* relationship involving two concrete instances and an abstract instance identifying the minimum behavioral requirements of the system for both concrete components.

For concrete instances C_1 and C_2, and abstract instance A, we define the behavioral substitutability relationship as follows:

$$\textbf{is\_sub}(C_2, C_1, A) \equiv C_1 \textbf{ implements } A \ \wedge \ C_2 \textbf{ implements } A \qquad (3.1)$$

This relationship may be read as "C_2 is substitutable for C_1 with respect to A". Although the behavior implemented by C_1 and C_2 may differ in a variety of ways, both components provide the behavior specified by A (assuming the **implements** relationships are justified). For a concrete template that **needs** A, either C_1 or C_2 will satisfy the requirement.

As an example, consider again Figure 3.22 on page 73. Given the relationships shown in this figure, `CI_Flipflop_1` is substitutable for `CI_Flipflop_2` with respect to `AI_Flipflop` and conversely, `CI_Flipflop_2` is substitutable for `CI_Flipflop_1` with respect to `AI_Flipflop`. Any two of `CI_FFWSet_1`, `CI_FFWSet_2`, and `CI_FFWSet_3`, are substitutable with respect to `AI_FFWSet`. Furthermore, any two of these three extension implementations are substitutable with respect to `AI_Flipflop` since each **implements** `AI_Flipflop`. Finally, each of the three extension implementations are substitutable for either of `CI_Flipflop_1` or `CI_Flipflop_2` with respect to `AI_Flipflop`. However, `CI_Flipflop_1` is not substitutable for `CI_FFWSet_1` with respect to `AI_FFWSet` since `CI_Flipflop_1` does not implement `AI_FFWSet`.

3.7 Chapter Summary

In this chapter we built on the more formal relations developed in Chapter 2 to define a useful set of component relationships. Section 3.1 introduced a notation based on RESOLVE and Ada for encoding specification and implementation components. The **uses** relationship, defined in Section 3.2, records any form of fixed dependency of one component upon another. Any type of component (abstract or concrete, template or instance) may use any other type of component. If component C_1 **uses** component C_2, then C_1 either refers directly or indirectly to C_2 (possibly implicitly) for the purpose of describing C_1's behavior in terms of the behavior described by C_2. Since the behavior described by a component will, in general, be influenced by any other component that it **uses**, clearly documenting this relationship is important for software maintenance.

The **implements** relationship, defined in Section 3.3, records the conformance of an implementation component to a specification component. Recording this relationship is useful for establishing substitutability properties, for stating its requirements for verification of correctness, and for documenting its "advertised" behavior. The **imps** relation, in terms of which **implements** is defined, is a relation defined over the sets of concrete instances and abstract instances. We have defined **implements**, however, as three related, but distinct, relationships. The three signatures of the **implements** relationships, listed in the left column of Table 3.1, correspond to: a concrete instance that **implements** an abstract instance, a concrete template that **implements** an abstract instance, and a concrete template that **implements** an abstract template, respectively.

The **extends** relationship, defined in Section 3.5, records the conformance of one abstract component to another. Recording the **extends** relationship is useful for establishing substitutability properties and for specifying the behavior of one component in terms of another component. Like **implements**, the name **extends** applies to three related, but distinct, relationships. The three signatures of the **extends** relationships, listed in the left column of Table 3.1, correspond to: an abstract instance that **extends** another abstract instance, an abstract template that **extends** an abstract instance, and an abstract template that **extends** another abstract template, respectively. In Section 3.5.2, we described the direct, coupled, and layered approaches to implementing an extension component and provided examples of each approach.

The **needs** relationship, defined in Section 3.4, records a behavioral requirement of a component as a deferred or polymorphic dependency. Using the **needs** relationship to express requirements, prevents unnecessarily coupling implementations and lays the foundation for improvements through component substitution. Used in

implements	concrete component C **implements** abstract component A iff C provides an implementation of all behavior specified by A
$CI \times AI$	**imps**(C, A) holds
$CT \times AI$	for *any* instantiation C' of C, **imps**(C', A) holds
$CT \times AT$	for *any* instantiation C' of C, there exists *some* instantiation A' of A, such that **imps**(C', A') holds.
extends	abstract component A_2 **extends** abstract component A_1 iff every concrete component that **implements** A_2 also **implements** A_1
$AI \times AI$	**exts**(A_2, A_1) holds
$AT \times AI$	for *any* instantiation A_2' of A_2, **exts**(A_2', A_1) holds
$AT \times AT$	for *any* instantiation A_2' of A_2, there exists some instantiation A_1' of A_1 such that **exts**(A_2', A_1') holds
uses	component C_1 **uses** component C_2 iff the meaning of C_1 depends either directly or indirectly on the meaning of C_2
needs	concrete template C **needs** abstract instance A iff C **uses** A and for all instantiations of C, C's references to elements in A are replaced by references to the corresponding elements in some concrete instance that **implements** A
is_sub	concrete instance C_2 is behaviorally substitutable for C_1 with respect to abstract instance A (**is_sub**(C_2, C_1, A)) iff C_1 **implements** A and C_2 **implements** A

Table 3.1: Summary of Component Relationships

conjunction with the **implements** relationship, the **needs** relationship isolates implementation components from each other prior to system integration and encourages the development of modularly verifiable components.

Finally, in Section 3.6, we defined the **is_sub** relationships which holds when two concrete instances are substitutable for one another with respect to a common specification — an abstract instance. Designing, implementing, and documenting software components using the **implements**, **extends**, **needs**, and when necessary, the **uses** relationships, is an important step toward component-level maintenance of software systems.

CHAPTER 4

PROGRAMMING LANGUAGE SUPPORT FOR BEHAVIORAL RELATIONSHIPS

In this chapter we examine how the component relationships described in Chapter 3 may be encoded in modern programming languages. We primarily focus on programming languages such as Ada and C++ which have an established user base and are generally regarded as useful for constructing large component-based software systems. Complex programming languages such as these provide many mechanisms which make possible a variety of approaches to encoding software components. The language mechanisms of primary interest are those supporting techniques for achieving *modularity*, *information hiding*, *polymorphism*, and *extendibility*. These four aspects of software engineering *roughly* correspond to the benefits associated with use of the **uses**, **implements**, **needs**, and **extends** relationships, respectively.

Section 4.1 begins this chapter with a review of the goals of an approaches to modularity, information hiding, polymorphism, and extendibility. The following sections discuss how programming language mechanisms may be used to encode the **uses**, **implements**, **extends** and **needs** relationships. Section 4.6 summarizes this chapter.

4.1 Language Support for Component-Based Software Engineering

Many authors have written about how programming languages can provide support for building reusable software components. Most detailed discussions of language mechanisms supporting component-based software focus on the features of a single language such as Ada [Boo87], C++ [CE95], Eiffel [Mey94] and RESOLVE [Har90]. Some books on object-oriented programming languages (OOPL's) compare how the mechanisms of different OOPL's support software reuse (for example, [Cox86, Bud91]). Edwards provides a detailed analysis of how well four languages — OBJ, RESOLVE, Eiffel, and Standard ML — support component-based software engineering [Edw95, pp. 165–183]. There is a wide variety of opinions about what

combination of specific language mechanisms best supports component-based software engineering. Nevertheless, there is general agreement that language mechanisms that support modularity, information hiding, polymorphism, and extendibility are particularly useful.

4.1.1 Modularity

Modularity in the design and implementation of software involves partitioning a software system into constituent "parts" — modules. The benefits of modularity are well established. Modular design is a primary tool for managing complexity using abstraction. Decomposing a large system into smaller, conceptually simpler units makes a system specification easier to understand and implement. Compared to a monolithic implementation, a modular implementation should be easier to understand, test, debug, and maintain. Furthermore, if modules are well-designed, the modular approach supports software reuse.

Component-based software engineering assumes that complex systems will be constructed from software components — modules. Programming languages supporting this approach must therefore provide some unit of modularity for defining components. The top-down or "structured" approach to analysis and design of software systems became popular in the 1970's [SMC74]. This approach advocates *functional decomposition* of systems. Functional decomposition focuses on *process abstraction* and leads to operations as the primary unit of modularity. With this approach, individual operations serve as components and component libraries primarily consist of collections of subroutines.

Object-oriented analysis and design, which began to gain popularity in the 1980's, takes a different approach to modularity which leads to different kinds of components [Par72, Mey87, Boo91]. The object-oriented approach focuses on *data abstraction* and decomposition of systems based on data structures rather than functionality. The primary rationale for the object-oriented approach is based on observation of how most large software systems change over time. The data structures of systems, when viewed abstractly, tend to be fairly stable over time. System functionality, however, tends to change to a much greater extent. The object-oriented approach views a system component primarily as an abstract data type (ADT) which specifies a type, operations on that type, and local state for representing the value of objects of the type. With this approach, components may contain data structures and multiple operations.

Many modern programming languages provide mechanisms for encoding components that encapsulate both data structures and operations. In OOPL's such as Simula, C++, Eiffel, and Java the primary unit of modularity is the *class*. A class serves double duty as a mechanism for both encapsulation and definition of user-defined types. C++ and Eiffel support parameterized classes (templates) which are

very useful for encoding the relationships described in Chapter 3. In other languages such as Ada, Modula-2, and ML (all strongly typed languages) the mechanisms for defining modules and types are distinct. An Ada *package*, a Modula-2 *module*, and an ML *module* may declare multiple types accessible by other components in addition to operations and local data structures[16]. Both Ada and ML provide strong support for parameterized components with *generic packages* and *functors*, respectively.

4.1.2 Information Hiding

Information hiding, also called *encapsulation*, is a technique for achieving abstraction whereby some features of a component are made inaccessible to (they are "hidden" from) other components. As with modularity, the benefits of information hiding are well known and most programming languages provide mechanisms supporting some form of information hiding. Information hiding may be used to restrict and simplify the way in which clients may interact with implementation components. By preventing client access to implementation details such as data representations, implementations may be changed without changing the "non-hidden" interface of a component. This reduces coupling between implementation components, supports making localized changes without global affects, and results in software systems which are easier to maintain.

In addition to simplifying a client's view of a component, information hiding used in conjunction with behavioral specifications may support *re-conceptualization* of a software component. That is, the "cover story" provided by a behavioral interface description (a specification component) might be quite different from the description provided by implementations of the specification. As a simple example, consider `AI_Flipflop` (Figure 3.1) used as the specification of `CI_Flipflop_2` (Figure 3.2). `AI_Flipflop` not only hides the `INTEGER` model representation of `CI_Flipflop_2`, it portrays — we might even say "lies about" — the implementation as having a `BOOLEAN` model. In this case the `BOOLEAN` model serves as a simpler, more abstract cover story for the actual representation of `CI_Flipflop_2`. One of the goals in the design of behavioral interface specifications is to convey to clients a useful *mental model* of the behavior exhibited by conforming implementations [Edw95, pp. 7–12]. In order to achieve simplicity and allow for a variety of differing implementations, the mental model described by a specification component may be significantly different from the model of any particular conforming implementation.

Programming languages provide a wide variety of tools for achieving information hiding at the component level. A common approach is to declare certain features of a component as "public" and others as "private". Types, operations, and variables

[16]Although Ada95 packages may export multiple types, only a single extensible "tagged" type may be defined within a package and extended using inheritance. With this limitation, the object-oriented model supported by Ada95 is similar to that of C++.

which are public may be directly referenced by (are visible to) client code. Features declared as private are only visible within the component in which they are defined thus achieving information hiding. In OOPL's, the data representation for the type defined by a class is typically represented by *instance variables* of the class. If the instance variables are declared as private, then the data representation of the class is hidden. C++ classes may define operations and variables with the three visibility modifiers: `public`, `private`, and `protected`. A variable or operation declared as `private` only is visible within the defining class. A `protected` operation or variable only is visible within the defining class and all sub-classes (classes linked by inheritance) of the defining class. Java has these three visibility categories plus two additional ones[17]. In Eiffel, all class features are private unless they are explicitly declared as "exported" in which case they are public.

As noted above, Modula-2 modules, Ada packages, and ML modules may define and export multiple types unlike classes in most OOPL's. In these languages, ADT's correspond to exported types with hidden data representations. Modula-2 modules and Ada packages consist of separate header and body parts, typically placed in separate files. The representation of a Modula-2 *opaque type* is hidden by declaring its visible representation as a pointer in the *definition module*, the header part, and declaring the referenced data structure in the private *implementation module*, the body part. An Ada *private type* is declared twice in an Ada *package specification*, the header part. A private type is first declared in the public section without providing a representation, and then in the private section with its representation. Although a client looking at an Ada package specification can "see" a private type's representation, client code does not have visibility to any types, operations, or variables declared in the private section of a package specification. Types, operations, and variables declared and implemented in the *package body*, but not declared in the public section of the package specification are completely hidden from clients of the package.

ML provides at least four different approaches to information hiding at the module level [Ull95, p. 163]. The approach of using ML *signatures* to hide information contained in ML *structures* corresponds very closely to using specification components to hide information contained in implementation components as shown in the examples in Chapter 3 and modeled by ACTI [Edw95, §4.13.3]. ML signatures are modules that only may contain type names (with no representation), value names with their associated type signature (but no value), and various other specifications. In ML, functions are treated as values. A value name and type signature may either be an ordinary variable and its data type or the name of an operation and its parameter profile. A structure is a module that may contain type representations, values including function implementations, and various other elements including nested substructures. Thus a signature module corresponds to a specification component

[17]Java's `private protected` is equivalent to `protected` in C++. Java's `protected` and `default` visibility take into consideration the Java "package" in which the classes are defined.

(with no language support for behavioral specifications) and a structure module corresponds to an implementation component. We discuss the relationship between ML signatures and structures further in Section 4.3.

4.1.3 Polymorphism

Polymorphism literally means "many forms". Within the context of programming languages, polymorphism refers to the situation in which a single name, such as a variable name, may be used to denote values of different types or objects of different classes. Cardelli and Wegner survey and classify a variety of techniques for achieving polymorphism [CW85]. They identify the two primary kinds of polymorphism as *parametric polymorphism* and *inclusion polymorphism*. Parametric polymorphism is achieved by using templates and inclusion polymorphism, also called *subtype polymorphism*, is achieved by using inheritance and typically dynamic binding of operations. Note that the term "polymorphism" is frequently used specifically to refer to subtype polymorphism with dynamic binding of operations, especially within the object-oriented community. As do Cardelli and Wegner, we use the term in the broader sense to include parametric polymorphism.

Techniques for achieving polymorphism support the design and implementation of components which are less coupled to other components than would be possible without polymorphism. This can help in attaining system maintainability, component reusability, and component substitutability. Budd nicely summarizes the role of polymorphism as follows.

> Polymorphism in programming languages permits the programmer to generate high-level reusable components that can be tailored to fit different applications by changing their low level parts. [Bud91, p. 88]

This characterization of polymorphism also describes the role of the **needs** relationship between a concrete template and an abstract component as discussed in Section 3.4. Consider the implementation `CT_Threeway_1` shown in Figure 3.8. `CT_Threeway_1` provides an example of parametric polymorphism. Since the representation of the concrete type `Threeway` is constructed from an implementation of `AI_Flipflop` supplied as a parameter, `Threeway` may be considered as a *polymorphic type* describing many different implementations (forms). The different implementations are all of the possible instantiations of `CT_Threeway`. The operations `Advance` and `On` defined by `CT_Threeway_1` may be considered as *polymorphic operations* which manipulate (flip-flop) objects of many different types. As we discuss in Section 4.5, it is also possible to encode the **needs** relationship using subtype polymorphism.

By the generally accepted definition of an OOPL, all OOPL's include inheritance and dynamic binding and thus the mechanisms necessary for subtype polymorphism. Statically typed OOPL's such as C++, Ada (Ada95), Java, and Eiffel as well as dynamic type checking OOPL's such as Smalltalk all support polymorphism in the form

of subtype polymorphism. Parametric polymorphism is supported in a variety of languages. Parameterized components may be encoded using ML functors, Ada generic packages, C++ templates, and Eiffel generic classes. Note that Ada, C++, and Eiffel (and several other, primarily research, languages) provide support for both subtype and parametric polymorphism. The mechanisms supporting these two forms of polymorphism may be used in combination to achieve useful results. We provide examples of the combined use of parametric polymorphism and inheritance in Chapter 5.

4.1.4 Extendibility

Extendibility refers to how easy it is to extend the functional behavior of an existing component-based system and thus how easy it is to extend the behavior described by software components. In Section 2.1.1 we discussed the inevitable need for changes to software systems. Perfective maintenance, which addresses changes in functional requirements, accounts for the largest portion of all maintenance costs [LBSB80]. Since new functional requirements nearly always call for additional system functionality (as opposed to reduced system functionality), extendibility is a highly valued characteristic of software systems.

Any software for which the source code is available is "extendable" in the sense that the source code can be modified to add new functionality. However, the goal of designing and implementing extendable components is to be able to extend the functionality of existing components with minimal disruption to systems that use those components. When extending the functionality of an existing component or component-based system, we want to modify the code of existing components as little as possible. Minimizing changes to existing component code minimizes: introduction of bugs and unexpected behavior, retesting and recertification, and possibly expensive system re-builds (e.g., extensive recompilation).

To some extent, there is a trade off between information hiding and extendibility. Extending a component that makes its internal details accessible (public) may be easier and result in a more efficient implementation of the extended functionality. However, as discussed in Section 3.5.2, there is a price to pay for such weak encapsulation. A number of authors have discussed this trade-off in the context of OOPL's [SG95, MW90, Sny86]. The layered approach to implementation extension described in Section 3.5.2 provides an example of how strong encapsulation may be maintained while still supporting extendibility.

In general, language mechanisms supporting modularity, information hiding, and polymorphism are also useful for supporting extendibility. The language mechanism most associated with extendibility is inheritance which is provided in one form or another by all OOPL's. Inheritance is a convenient mechanism for describing how one component differs from another. However, inheritance is used for many different purposes in addition to extension of components. In presenting a taxonomy of the

uses of inheritance, Meyer describes 12 distinct "valid" uses for inheritance [Mey96]. Under the general category of *model inheritance*, Meyer includes *subtype inheritance* which does not involve inheritance of data representations or operation implementations. Subtype inheritance is used to express and enforce a conformance relationship. We refer to this use of inheritance as *specification inheritance.* The most common use of inheritance involves inheritance of data representations and operation implementations. This use of inheritance is often called *subclassing.* We refer to any use of inheritance that falls into this category as *implementation inheritance.* What Meyer calls "extension inheritance", a **uses** and subtyping relationship between implementations, and "implementation inheritance", a **uses** relationship based on inheritance but not implying subtyping, both fall into this category.

Many authors have written about the problems that can arise from using one programming language mechanism, inheritance, for several distinct purposes [Tai96, Cla95, Edw93, Coo90]. Several newer OOPL's address these problems by using distinct language mechanisms for encoding (structural aspects of) specification inheritance and implementation inheritance. Java is the most widely used language that has different mechanisms for specification inheritance and implementation inheritance. We discuss these aspects of Java in Sections 4.3.4 and 4.4. Other new OOPL's that also use different language mechanisms for specification and implementation inheritance include Theta, a language primarily based on CLU [LCD+94], Sather 1.0, a language based on Eiffel [SOM94], and Pizza, a superset of Java [OW97]. Note that unlike Java, Theta, Sather, and Pizza all support parametric polymorphism in addition to inclusion polymorphism.

The *hierarchical library* structure introduced into Ada with the 1995 language revision provides a unique approach to component extension not found in other programming languages. In Ada, any library unit (a package, subprogram, or generic unit) may be extended by a *child unit.* The visibility of a child unit includes full visibility of its parent unit including the parent's private section. The presence of a child unit, however, does not affect the parent unit or any components which depend on the parent unit. Child and parent units may be compiled separately and adding a child unit does not require recompilation of the parent unit. The child unit mechanism is orthogonal to Ada's inheritance mechanism. However, as we demonstrate in Chapter 5, these two language mechanisms may be used in combination.

4.2 Encoding The uses Relationship

As discussed in Section 3.2, the **uses** relationship represents a fixed dependency between two software components. If component X **uses** component Y, then X in some way depends on Y. The **uses** relationship provides no information on *how* Y is used by X; but without access to component Y, the meaning of component X is incomplete. With some programming languages, the source or object code of Y must

be available in order to compile the source code of X. Other languages, however, may not require that the code of Y be available to X until system integration time (link time) or even until runtime.

Some programming languages require explicit encoding of the **uses** relationship between components. The visibility rules of the language largely dictate what is required for one component to refer to elements defined by another component. In the examples in Chapter 3, a `uses` clause in `context` section directly encoded the **uses** relationship. In the case where X **uses** Y, we assumed, for simplicity, that all elements defined in the interface of Y could be referenced directly within X without name qualification (unless necessary to resolve overloading). In some languages there is a separate mechanism for allowing abbreviated reference to components and elements defined within other components.

In some cases, a `uses` clause in the Chapter 3 examples conveys redundant information that may be deduced from other parts of the component. For example, since the `needs` clause always entails a **uses** relationship we could have omitted all `uses` clauses for components subsequently included in `needs` clauses. Also, `uses` clauses could be omitted for each component mentioned in a `re-exports` clause. We choose to require a `uses` clause for each **uses** relationship to make it perfectly clear in the source code on what other components a given component directly depends. This information is critical for component understanding, maintenance, and program analysis, and should not be obscured in any way.

One characteristic that further distinguishes among programming languages supporting component-based software engineering is the distinction between components and data types. In Ada and languages based on Niklaus Wirth's Modula (Modula-2, Modula-3, Oberon, and Oberon-2 [Wir82, RW92]), a component (a package or module) is *not* a data type. In these languages, the **uses** relationship is explicitly encoded. In most OOPL's[18], however, a component, typically called a class, is a user-defined type from which objects may be declared. Partly as a result of this distinction, most OOPL's do not require a class to explicitly list all other classes that it directly **uses**.

In Ada, a `with` clause placed in the context section of a package encodes the direct **uses** relationship between two packages. If an Ada package, say package X, needs to refer to some element defined in another package, say package Y, then X must include a `with` clause naming Y[19]. In general, an Ada package only may be compiled when all packages upon which it depends have been compiled and are available in the program library. This strategy helps ensure that structural errors are detected as early as

[18] Ada (Ada95), Modula-3, and Oberon all have language mechanisms supporting object-oriented programming. The support these languages provided for modularity has been carried over from their non-object-oriented precursors, Ada83 for Ada and Modula-2 for Oberon and Modula-3.

[19] All Ada packages implicitly have visibility to the special package `Standard` which defines Ada's built-in types and operations. Also, packages defining child units (discussed in Chapter 5) have visibility to their parent unit but do not require a `with` with clause naming their parent. Instead, the parent unit name is a prefix of the child unit name.

possible. Note that Ada's `use` clause allows components to reference public elements defined in `with`'ed packages without using their fully qualified names. Chapter 5 includes examples showing the use of Ada's `with` and `use` clauses.

In the Modula and Oberon family of languages, the `IMPORT` clause "imports" (makes visible) elements from another module in much the same manner as Ada's `with` clause. In Modula-2 and Modula-3 it is possible to selectively import features exported by another module by using a clause of the form "`FROM M IMPORT X`" where `M` is a module name and `X` is an explicitly exported (public) element defined in `M`. A module that includes this form of `IMPORT` clause may refer to the imported element `X` directly instead of using the qualified name `M.X`. Oberon does not have this form of `IMPORT` clause since its designer believed that explicit qualification of imported names is preferable, especially when many modules are involved [RW92].

In typed OOPL's such as Java and Eiffel, there are two ways in which one class may have a fixed dependency on another. First, class X may be a *subclass* of class Y, in which case X **uses** Y. This form of dependency is encoded using an inheritance mechanism such as the `inherit` clause in Eiffel and the `implements` and `extends` clauses in Java. Second, X may be a client of Y without being a subclass of Y. In this case, X uses the name Y as a *data type* for declaring an object or parameter. (In Java, Y could also be used within X for type casting.) When X is a client, but not a subclass of Y, most OOPL's do not require any sort of "import list" that in a single place names all fixed dependencies on other components. Meyer, in describing Eiffel [Mey88, p. 211], and Stroustrup, in describing C++ [Str93, p. 416], both note that such an import list, would be redundant and could be automatically generated by a tool. Without the aid of such a tool, however, a maintainer must search for class names throughout a given class in order to determine all inter-component dependencies.

To avoid possible confusion, we note that Java does have an `import` statement. However, the purpose of Java's `import` statement is to allow a class to refer to other classes using abbreviated names rather than fully qualified names. Thus Java's `import` statement serves essentially the same role as Ada's `use` clause.

Like most other OOPL's, C++ does not directly support encoding of the **uses** relationship between a class and the other classes of which it is a client but not a subclass. However, a common C++ idiom is to use the preprocessor directive `#include` to textually insert a C++ header file containing a class interface into another file that uses the interface. If, by discipline, each class is associated with a single header file that declares its interface, then `#include` may be used to encode the **uses** relationship between two classes. If client class X `#include`'s the header file for class Y, and we assume that at link time X will get linked to the class definition for Y, then it is reasonable to consider the `#include` directive a direct encoding of the **uses** relationship. Note that in the case of specification components represented as abstract classes, there need not be an associated class definition to a class header file.

To summarize, in most OOPL's a module is a class which is a data type. In this case, the set of fixed dependencies of component X corresponds to all data types referenced within X excepting the type X itself and certain built-in types. These languages do not require explicit encoding of the **uses** relationship and a tool may be necessary to identify and summarize such dependencies. Languages such as Ada and the Modula family of languages, in which the mechanisms for defining components and data types are distinct, provide direct support for encoding the **uses** relationship in the form of an import list.

4.3 Encoding The implements Relationship

In Section 3.3 we defined the **implements** relationship which is based on the **imps** relation defined in Section 2.3.1. The motivation for establishing and clearly documenting the **implements** relationship was discussed in these sections and also in Section 2.1. In this section we discuss how various programming language mechanisms may be used to encode the *claim*[20] that the **implements** relationship holds between two components — an implementation and a specification.

The primary reasons for explicitly encoding the **implements** relationship are:

- to provide information used to determine appropriate component composition,
- to indicate an obligation for conformance checking, and
- to help document the claimed behavior of implementation components.

At component integration time, a record of the **implements** relationship may be used by a linker or other tool to determine which component compositions are appropriate and which are not. For example, if concrete component X **needs** abstract component Y and concrete component Z **implements** Y, then it is appropriate for X to be instantiated with Z. An encoded **implements** relationship also may require a compiler to check structural conformance between the related implementation and specification. Similarly, the **implements** relationship may generate proof obligations for a verification tool or testing requirements in order to aid in confirming behavioral conformance between two components. By associating an implementation with a specification to which it must conform, the **implements** relationship also serves as documentation useful to a software engineer working directly with the source code of an implementation.

[20]For brevity, we will typically use the phrase "encoding an **implements** relationship" to mean more accurately "encoding *the claim of* an **implements** relationship".

4.3.1 The implements Relationship and Coupling

Recall from Chapter 3 the components `CI_Flipflop_2` and `CI_Flipflop_3` (Figures 3.2 and 3.5, respectively). Both components implement `AI_Flipflop` (Figure 3.1) and both describe identical operational behavior. The difference between these two components is that `CI_Flipflop_3` **uses** `AI_Flipflop` while `CI_Flipflop_2` does not. There is no reason, in general, why a concrete component must *depend on* an abstract component which it **implements**. However, many of the language mechanisms most useful for associating specifications and implementations couple implementation components to the specification components which they implement. Thus in most languages, encoding the relationship C **implements** A requires that C **uses** A. We discuss a few interesting exceptions to this in Section 4.3.5.

There are several reasons why it is useful for an implementation component to depend on a specification component which it implements. Stating a (claimed) **implements** relationship in the source code serves as documentation identifying a specification of the component's implemented behavior (although not necessarily all of the implemented behavior). For documentation only purposes, however, an implements statement may be treated as a semantically irrelevant comment ignored by compilers and other processing tools. That is, there need be no syntactic or semantic dependency just to achieve this documentation objective, and thus no **uses** relationship between the two components.

As with `CI_Flipflop_3`, a concrete component may refer to specification (non-programming) elements of an abstract component which it implements. Assuming no renaming of types, operations, or variables (programming elements) defined in the abstract component, this level of reference is sufficient for expressing the correspondence (abstraction relation). As we discussed in Section 3.3, recording the correspondence is an important aid to justifying the **implements** relationship. In this case, the implementation **uses** the specification, but not in the normal "compilation dependency" sense. A typical compiler could process such an implementation component without examining the implemented specification component. In RESOLVE/Ada95, discussed in Chapter 5, specification elements are encoded as comments which are ignored by Ada compilers.

One of the main reasons programming languages require implementation components to be coupled to interface specification components is to support structural conformance checking. That is, language mechanisms useful for encoding the **implements** relationship — most notably inheritance mechanisms — typically require the compiler to check that the structure of the implementation conforms to that of the specification. We discuss conformance checking below in Section 4.3.2.

Despite these reasons for making an implementation dependent upon the specification(s) that it **implements**, there are some potential disadvantages to this common approach. For example, if the content of a concrete component must explicitly name any abstract components that it **implements**, then establishing a new **implements**

relationship (not derivable from existing relationships) for an existing concrete component will typically require modifying the implementation's source code and potentially expensive recompilation. A few languages such as ML and C++ with signatures [BR97] provide support for encoding the **implements** relationship without requiring that implementations refer to the specification components they implement. We consider an example of this in Section 4.3.5.

4.3.2 Conformance Checking

Prior to placing a concrete component into a component library and making it available for client use, we obviously want to have some confidence that it will behave as "advertised" when integrated into a system. A component's advertised behavior is the behavior specified by the abstract components which it **implements**. Conformance checking is the process of determining — to some level of confidence — that a concrete component correctly describes an implementation of the behavior specified by an abstract component which it **implements**. In order for an **implements** relationship to be justified, the structure *and* behavior of the concrete component must conform to that of the abstract component.

For most programming languages, checking structural conformance is a relatively straightforward task carried out by a compiler or interpreter. This involves ensuring that all types, operations, and variables specified in the abstract component are matched by compatible concrete types, operations, and variables in the (client-visible part of) the concrete component. The type system of the language determines the rules for conformance. For languages that support type extension (inclusion polymorphism) and parameterized types (parametric polymorphism), the rules for determining what constitutes a match can become somewhat complex [CW85, LW94].

As we have discussed in earlier chapters, very few programming languages include mechanisms supporting specification of component behavior. Exceptions are primarily research languages such as OBJ [Gog84] and RESOLVE [SW94]. A more common approach for constructing behavioral interface specifications (abstract components) is to integrate the use of independent specification and implementation languages [SW94, DL96, Jon90, LvHKBO87]. In practice, however, the most common approaches for encoding the behavior specified by an interface rely on informal, nonrigorous descriptions of component behavior. Unfortunately, informal specifications are usually imprecise and ambiguous. Therefore, reasoning about the behavior of a concrete component which someone claims **implements** such a specification may be faulty.

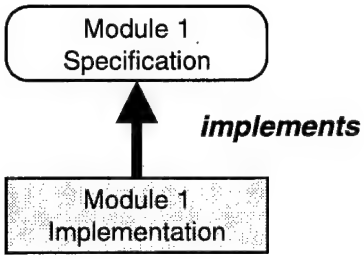


Figure 4.1: A One-To-One Implementation-To-Specification Relationship

4.3.3 One-to-One Relationships

One of the first programming languages designed specifically to support modular (component-based) software development was Modula-2 [Wir82]. In Modula-2, software components are encoded as modules which may be divided into two parts: a *definition module* and an *implementation module*. A definition module contains signatures of types and operations. An implementation module contains data structures and operation implementations. A definition module may be used to represent an abstract instance and an implementation module may be used to represent a concrete instance. The Modula-2 compiler checks to ensure that an implementation module structurally conforms to a definition module of the same name. Thus, in Modula-2 there is a one-to-one, by-name conformance relationship between implementation and specification modules. This relationship naturally serves to represent the **implements** relationship when other means are used to enforce behavioral conformance between the implementation and definition pair.

In Ada, a software component is typically encoded as a package. An Ada package has two parts: a package specification and a package body. These serve the same roles as Modula-2's definition and implementation modules, respectively. As with modules in Modula-2, there is a one-to-one, by-name conformance relationship between a package specification and a package implementation. Unlike Modula-2 modules, however, Ada packages may be generic (parameterized). Generic package specifications and package bodies may be used to encode template components. Figure 4.1 depicts the one-to-one **implements** relationship between a specification component, such as a Module-2 definition module or an Ada package specification, and an implementation component, such as a Modula-2 implementation module or an Ada package body. As depicted by the thick arrow, this is also a **uses** relationship since the implementation components have visibility over and depend upon (for compilation) their corresponding specification components.

The implementation-to-specification relationships encoded using Modula-2 modules and Ada packages help reduce the coupling between implementation components. For example, if implementation component C **needs** specification component A, then C may refer to A and be compiled without the corresponding implementation of A present. Further more, the concrete component that **implements** A may be modified and recompiled without requiring recompilation of client components which **use** A. However, if A is a Modula-2 definition module or an Ada package specification, a component library containing A may only have a single implementation component that encodes the **implements** relationship with A[21]. Thus, these language mechanisms, modules and packages, lack support for multiple implementations of a single specification. Furthermore, Modula-2 modules and Ada packages (prior to the 1995 language revision) are not easily extendable. The successors to Modula-2, Oberon [RW92] and Modula-3 [Har82], and the 1995 revision to Ada [Int95b] all include mechanisms that support multiple implementations and component extension.

4.3.4 Many-to-One Relationships

In Section 4.1 we discussed inheritance and classes, and stated that specification inheritance is useful for encoding the **implements** and **extends** relationships. One of the many uses of inheritance is the expression and enforcement of a conformance relationship between two data types. When a component specifies or implements a *single* data type, as does a class, inheritance may be used to express a conformance relationship between two components. For example, if class D is derived from (inherits from) class B, then class D is a subclass of B and, in most cases, exports all operations, variables, exceptions, etc., exported by B. Thus, class D structurally conforms to class B[22].

With specification inheritance, the base class, from which conforming subclasses are derived, does not provide any implementation detail. This is analogous to a Modula-2 definition module and (the public part of) an Ada package specification. In OOPL terms, a class which does not provide a type representation and implementations for all its operations is called an *abstract class* or an *abstract base class*. Most OOPL's have mechanisms for encoding abstract classes or their equivalent. In C++ a class with at least one *pure virtual function* is an abstract class. In Ada, a package exporting an **abstract type** corresponds to an abstract class. In Eiffel, an abstract class is called a *deferred class*.

[21] Various tools and tricks may be used to circumvent the language limitation of one package body per package specification within an Ada component library. However, within a single executable system, there may only be one package body per package specification.

[22] Some OOPL's allow a derived class D to "hide" inherited operations of base class B. If this technique is used, then D will not conform to B.

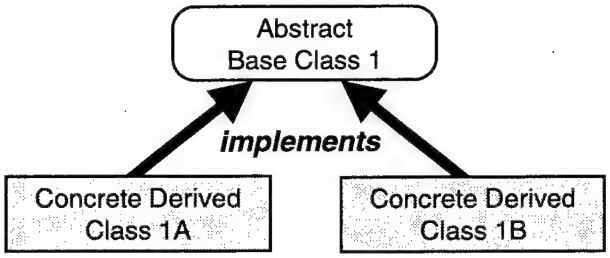


Figure 4.2: A Many-To-One Implementation-To-Specification Relationship

An abstract class that provides no implementation detail is useful for encoding an abstract component that exports a single type. A *concrete class*, a class that fully implements its data representations and operations, is useful for encoding a concrete component that exports a single type. The structural aspects of the **implements** relationship may be encoded using specification inheritance by deriving a concrete class from an abstract class.

Figure 4.2 depicts two (claimed) **implements** relationships among two concrete classes and an abstract class. Each concrete class is derived from the abstract class shown at the top of the figure. Although a language's inheritance mechanism may be used to encode this type derivation, the concrete components need not "inherit" anything from the abstract class. Each concrete class must override all abstract operations specified by the abstract class with structurally conforming and fully implemented operations. Additionally, each concrete class must provide a data structure to represent the exported type of the concrete class. Note that when specification inheritance is used to encode the **implements** relationship, the implementation component **uses** the specification component that it claims to implement. This is indicated by the thick arrows in Figure 4.2.

In contrast to the one-to-one implementation-to-specification relationship depicted in Figure 4.1, inheritance supports encoding many-to-one **implements** relationships. Any number of concrete classes may be derived from and implement a single abstract class. Using this approach, a component library, and even a single executable program, may include different concrete components, each of which explicitly **implements** a single abstract component. Thus this use of inheritance supports (better than the mechanisms of Modula-2 and Ada83) encoding the **implements** relationship and component-level maintenance. The RA95 discipline (discussed in Chapter 5) and RESOLVE/C++ discipline both use specification inheritance for encoding the **implements** relationship.

In Section 4.1.4 we pointed out that some newer languages use different mechanisms for specification inheritance and implementation inheritance. Currently, the

most popular of these languages is Java. In addition to classes which may be abstract or concrete, Java includes *interfaces* which are similar to abstract classes. Unlike Java's abstract classes, however, interfaces may only include abstract operations (called abstract methods in Java) and constants. Java interfaces define structural component interfaces and serve as a basis for specification inheritance.

Java's keyword `implements` is used for specification inheritance only while the keyword `extends` may be used for specification or implementation inheritance[23]. In a Java class or interface, the `extends` clause encodes the traditional form of inheritance found in many OOPL's. However, a class may only "`extend`" one other class. This limits Java's implementation inheritance to *single inheritance*. On the other hand, a Java class may "`implement`" one or more interfaces and a Java interface may "`extend`" one or more interfaces. Thus Java's specification inheritance, encoded by an `extends` clause in an interface or an `implements` clause in a class, supports *multiple inheritance*. Multiple inheritance, which often leads to problems when used for implementation inheritance, does not cause the same problems when used for specification inheritance. Java's use of distinct language mechanisms to support these two different, and sometimes conflicting, uses of inheritance is an improvement over the more traditional approach of using one inheritance mechanism for both purposes.

In Java, an abstract instance exporting a single type may be encoded by an interface. A concrete instance exporting a single type may be encoded by a concrete class (a class with no abstract methods). Using this strategy, the structural aspects of the **implements** relationship may be encoded conveniently with an `implements` clause in a concrete class that names an interface, which the concrete class **implements**. Figure 4.3 shows the Java interface `AI_Flipflop` followed by the Java concrete class `CI_Flipflop_3`. These are the Java encodings of the abstract instance shown in Figure 3.1 and the concrete instance shown in Figure 3.5. Note that the parameter to `Toggle` and `Test` (of the exported type) is implicit in Figure 4.3 since Java uses the traditional object-oriented notation for method declarations and invocations.

Despite Java's support for conveniently encoding the structural aspects of the **implements** relationship, as shown in Figure 4.3, Java lacks support for encoding a more general **implements** relationship. A Java class only can define a single extendable type. Thus, a Java class cannot define two related types, such as `Point` and `Line`, both of which may be extended using inheritance. This limitation is common to the object-oriented paradigm, which uses a single mechanism, typically a class, to define both components and programmer-defined ADT's. Mechanisms such as Java *packages* and C++ *friend functions* provide inelegant means for working around this limitation in most situations.

Java's lack of support for parameterized classes, templates, presents a more serious limitation. Various idioms exist for "simulating" templates in Java [OW97].

[23]We coined the terms "implements" and "extends" before looking at Java. Their use by Java reinforces our belief that these are natural terms for conveying the intended relationships.

```
// AI_Flipflop.java

public interface AI_Flipflop {
   // modeled by BOOLEAN
   // exemplar ff
   // initially ff = FALSE

   public void Toggle ();
      // ensures ff = NOT #ff

   public boolean Test ();
      // ensures Test = ff
}


// CI_Flipflop_3.java

public class CI_Flipflop_3
   implements AI_Flipflop {

   private int state = 0;
      // convention 0 <= state <= 255
      // exemplar ff_rep
      // correspondence ff = ((ff_rep.state MOD 2) = 1)

   public void Toggle () {
      state = (state + 1) % 256;
   }

   public boolean Test () {
      return ((state % 2) == 1);
   }
}
```

Figure 4.3: Java Encoding of CI_Flipflop_3 **implements** AI_Flipflop

However, using such strategies to encode a template-to-template **implements** relationship leads to extremely awkward and often inefficient code. Some researchers have proposed the addition of templates to Java [BLM96, OW97].

4.3.5 Many-to-Many Relationships

As we noted in Section 2.3.1, the **imps** relation is a *many-to-many* relation. Therefore, the **implements** relationship is a many-to-many relationship in the following sense. Many different concrete components may implement a single abstract component; and a single concrete component may implement many different abstract components. The first case, multiple implementations of a single specification, is a

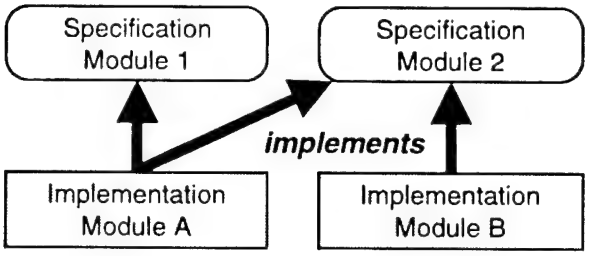


Figure 4.4: Many-To-Many Implementation-To-Specification Relationships

common situation and essential for component-level maintenance of component based systems (both software and hardware systems). The inheritance-based approach discussed in Section 4.3.4 supports multiple implementations of a single specification. The second case, multiple specifications of a single implementation, arises implicitly from specification extension (recall Equation 2.4) and also may be useful to provide distinct interfaces or "views" of a single implementation.

It is possible to explicitly encode **implements** relationships among a single concrete component and more than one abstract component. For example, in programming languages that support multiple inheritance, a concrete class may inherit from multiple abstract classes. In Java, a concrete class may include an `implements` clause that names more than one interface. In this situation, structurally identical methods may (but need not) be defined in and "inherited" from more than one interface. Figure 4.4 depicts a simple many-to-many implementation-to-specification scenario. In this example, the concrete component encoded as implementation module A **implements** *and* **uses** both abstract components, specification modules 1 and 2. Implementation module A must conform to both specification modules and may be used where implementations of either or both specifications are required. As in Figure 4.2, the **uses** relationship between implementation and specification comes from employing an inheritance mechanism to encode the **implements** relationship.

Encoding an **implements** claim with any mechanism that couples an implementation to a specification has some disadvantages. Inheritance is the primary example of such a mechanism. For example, to encode a new **implements** relationship for an existing concrete component, perhaps one in object code form in the component library, requires a source code change and recompilation of the concrete component. Baumgartner and Russo present an example of how this problem might arise in practice in the context of OOP [BR97, § 2.1]. The solution they propose and implement for C++ [BR97] and, with Läufer, for Java [LBR96] centers around *structural conformance* of components. With structural conformance, a class does not have to name

the interface[24] to which it conforms as is required with inheritance. At component integration time, the compiler determines whether a class conforms to an interface by comparing its structure (public names and signatures) to the structure of the signature. A similar structural comparison is performed by ML in determining whether an ML structure (an implementation component) conforms to an ML signature (a structural specification component).

With the structural conformance approach, an implementation component does not need to be coupled to a specification component that it implements in order for an instance of client code that **needs** the specification component to use the implementation component. Therefore, an existing implementation component does not need to be modified in order for it to fulfill new, and possibly unforeseen, requirements. Furthermore, this strategy clearly supports many-to-many **implements** relationships among concrete and abstract components.

As with the other approaches discussed in this chapter, language mechanisms supporting structural conformance do not address behavioral conformance between components. Furthermore, structural conformance does not require any language mechanisms to explicitly record the **implements** relationship as does inheritance. As a result, it is possible for two components to "accidentally" conform structurally but not conform behaviorally. Läufer et. al., propose the use of *properties* to address the problem of accidental conformance [LBR96, p. 6]. Properties are dummy methods with "well known names" that encode semantic information. For example, the property "LIFO" might be included in abstract and concrete classes for stack components.

A more powerful and flexible solution to this problem is to use independent mappings between components, which describe how the behavior of one component conforms to that of another component. As we discussed in Section 3.3, recording the correspondence (abstraction relation) between two components documents how the behavior described by one component may be understood and justified as conforming to the behavior described by another component. In the examples in Chapter 3 (Figures 3.5, 3.7, 3.8, 3.11, 3.18, and 3.27) the correspondence is recorded in concrete components and implicitly associated with the abstract component named in the `implements` clause. Furthermore, in each case there is an implicit mapping of each element of the abstract component to its corresponding implementation element in the concrete component. This approach couples a concrete component to the abstract component(s) that it **implements** and fixes the set of abstract components that may be used as client-level descriptions of the implementation. It also fixes the way in which the concrete component is interpreted as conforming to an abstract component. To add new relationships or modify existing ones requires modifying the

[24]In their proposal for C++, Baumgartner and Russo introduce a language construct called a *signature*, to which classes may conform structurally without inheritance. In their proposal for Java, Java interfaces serve this purpose.

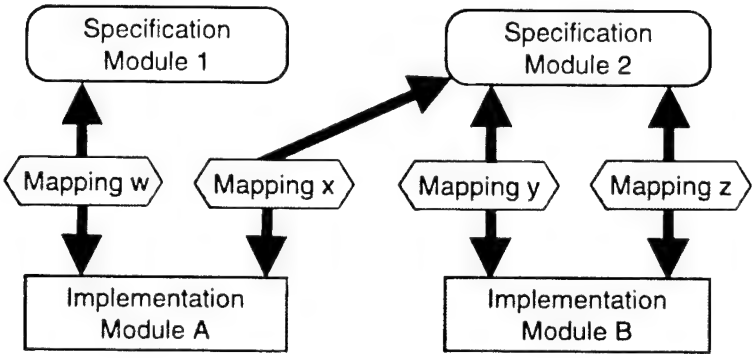


Figure 4.5: Independent Mappings Between Specifications and Implementations

concrete component even though the operational behavior the component implements does not change.

An alternative approach is to record conformance relationships, such as **implements** and **extends**, in a separate module. This approach is used by the functional language OBJ [Gog86] and is modeled by *interpretation mappings* in the ACTI model of software subsystems [Edw95, §4.10]. In OBJ, a module can be either an *object* or a *theory*. An OBJ object defines types (called *sorts* in OBJ) and associated operations. An OBJ theory provides an abstract, axiomatic description of behavior. Thus, OBJ objects and theories correspond to concrete and abstract components, respectively. An OBJ *view* describes how a module, either an object or a theory, conforms semantically to a theory. Thus, a view may be used to describe how an object **implements** a theory or how a theory **extends** another theory. In ACTI, an interpretation mapping plays a similar role. An interpretation mapping defines a correspondence between two abstract instances, explaining how one abstract instance can be interpreted as satisfying the behavior described by the other.

Figure 4.5 shows how two abstract components, `Specification Module 1` and 2, could be related to two concrete components, `Implementation Module A` and B, with four independent "mapping units". In this example, each of the four mapping units could describe a different **implements** relationship between the two components to which it refers. As the arrows in Figure 4.5 imply, the mapping units refer to (depend on) the specification and implementation modules, but not the other way around. Thus an implementation need not refer to a specification that it **implements** and vice versa. This strategy allows new explicit conformance relationships to be added to a component library without modifying existing components which may be involved in the new relationships.

Figure 4.5 shows two different mapping units, `Mapping y` and `Mapping z`, relating `Implementation Module B` and `Specification Module 2`. This is possible because a mapping unit can rename various elements in describing how once component conforms to another. Both OBJ views and ACTI interpretation mappings support renaming. As a more concrete example, assume that `Specification Module 2` in Figure 4.5 describes a stack abstraction such as `AT_Stack` shown in Figure 3.10 on page 57. Now assume that `Implementation Module B` describes an implementation of a deque, a double-ended queue, including the deque operations: `Enqueue_At_Front`, `Enqueue_At_Rear`, `Dequeue_At_Front`, and `Dequeue_At_Rear`. `Mapping y` could map `Enqueue_At_Front` and `Dequeue_At_Front` to the stack's `Push` and `Pop` operations, respectively. Describing a different way to implement a stack, `Mapping z` could map `Enqueue_At_Rear` and `Dequeue_At_Rear` to `Push` and `Pop`, respectively. To complete the picture, `Implementation Module A` might be at typical stack implementation with only operations `Push` and `Pop`. In this case `Mapping x` would describe the obvious mapping with no renaming. `Specification Module 1` might describe the behavior of a bag (multiset) container with operations `Insert` and `Remove`. Then `Mapping w` would map `Push` and `Pop` to `Insert` and `Remove` respectively.

The independent mapping approach provides significant flexibility. However, it adds complexity. The **implements** relationship is no longer a simple binary relationship as modeled by **imps** in Section 2.3.1. Also, when a mapping unit is allowed to rename types and operations, it plays an important operational role at component integration time. In addition to selecting an implementation component to instantiate a template, the system developer may also need to select an associated mapping unit to identify the appropriate operation-to-operation bindings. OBJ uses *default views* (default mappings) to simplify component composition [Gog86, p. 21].

4.4 Encoding The extends Relationships

In Section 3.5 we defined the **extends** relationship which is based on the **exts** relation defined in Section 4.1.4. Documenting the **extends** relationship records the claim that any implementation of one specification will also be an implementation of another specification. Thus if abstract component A_2 **extends** abstract component A_1, then all implementations of A_2 will also be implementations of A_1. Thus A_2 must specify all the behavior specified by A_1 and usually will describe additional functional behavior not specified by A_1. The motivation for establishing and clearly documenting the **extends** relationship is to foster the development of new components with enhanced capabilities, but which remain compatible with existing systems and their requirements.

In many respects, the **extends** relationship is similar to the **implements** relationship. Both are many-to-many behavioral conformance relationships. Many of the issues discussed in Section 4.3 pertaining to encoding the **implements** relationship

also apply to encoding the **extends** relationship. For example, while **extends** is not a dependency relationship, like **implements**, it is easiest to encode as such. Like **implements**, **extends** is defined as a many-to-many relationship. However, most language mechanisms useful for encoding **extends** are the same as those used for encoding **implements** and only support many-to-one **extends** relationships. We do not address these issues in detail in this section since they were discussed at length in Section 4.3.

The most common way to encode the **extends** relationship in modern programming languages is to use an inheritance or inheritance-like language mechanism. In Section 3.5.1 we discussed the differences between the meaning of **extends** and the effect typically achieved by using inheritance. The **extends** relationship holds between two specifications when one conforms to another. Neither of the two specifications need mention the other (as with `AI_FFExt` in Figure 3.15 which **extends** `AI_Flipflop` in Figure 3.1). Use of inheritance always implies a **uses** relationship. If abstract class A_2 inherits from abstract class A_1 (type A_2 is derived from type A_1) then A_2 **uses** A_1. As discussed in Section 3.5.1, there are advantages and disadvantages to using a coupling relationship to encode the **extends** relationship.

Several newer OOPL's such a Java provide distinct language mechanisms for adding additional elements to abstract "interface-only" components and for adding additional elements to implementation components. For example, Java's `extends` keyword, when used to relate two Java interface components, provides a form of specification inheritance. Figure 4.6 shows an example of this use of Java `extends`. The interface `AI_Flipflop_With_Set` shown in the bottom of this figure **extends** `AI_Flipflop` shown in the top of the figure. The operations `Toggle` and `Test` are inherited from `AI_Flipflop` by `AI_Flipflop_With_Set`.

Java's `extends` mechanism is convenient for encoding the **extends** relationship between abstract instances as shown in Figure 4.6. However, since Java does not support templates, only the abstract instance to abstract instance form of **extends** (defined on page 61) may be encoded in Java. Furthermore, when the `extends` keyword is used to relate two Java classes, it provides implementation inheritance rather than specification inheritance. While the common meaning of the term "extends" applies to both situations, we prefer to classify the use of `extends` between two classes as a specific application of the **uses** relationship. While implementation inheritance is useful for coding coupled implementations of extension components as discussed in Section 3.5.2, it is also commonly used in situations where there is no intended behavioral conformance between the components.

```
// AI_Flipflop.java

public interface AI_Flipflop {
   // modeled by BOOLEAN
   // exemplar ff
   // initially ff = FALSE

   public void Toggle ();
      // ensures ff = NOT #ff

   public boolean Test ();
      // ensures Test = ff
}


// AI_Flipflop_With_Set.java

public interface AI_Flipflop_With_Set
   extends AI_Flipflop {

   public void Set ();
      // ensures ff = TRUE
}
```

Figure 4.6: Java Encoding of AI_Flipflop_With_Set **extends** AI_Flipflop

Unlike Java, most OOPL's use the same inheritance notation for at least three conceptually distinct purposes: implementation to specification conformance, specification to specification conformance, and implementation to implementation conformance. Here we are assuming that abstract classes are used as specification components. In Chapter 5 we discuss how RA95 uses Ada's single inheritance mechanism for each of these three purposes.

In addition to inheritance, Ada also provides another mechanism which supports extension of components (packages). Ada's hierarchical library units allow a "child unit" package to extend an existing "parent unit" package without requiring any alterations to the parent unit or to systems which use the parent unit. (We discuss this mechanism in detail in Chapter 5.) While inheritance is a mechanism for extending a type, a child unit extends an Ada package which may include definitions of more than one type. Thus, the hierarchical library unit mechanism offers some advantages over inheritance and is indeed quite useful. However, this mechanism does not support substitutability of components as does inheritance. This is because there is no way to encode the **needs** relationship based on hierarchical library units. That is, there is no way in Ada to encode the requirement that *any* child unit of a parent unit may be supplied as an actual parameter to a concrete template (a generic package)

when the formal parameter is restricted to be an instance of the parent unit. Thus, the hierarchical library unit mechanism alone is not appropriate for encoding the **implements** relationship in a way that supports component-level maintenance.

4.5 Encoding The *needs* Relationship

In Section 3.4 we defined the **needs** relationship which is based on the **needs** relation defined in Section 2.4.2. The **needs** relationship expresses a deferred dependency on an implementation component. In this section we briefly discuss how the **needs** relationship may be encoded with programming languages.

In programming language terms, using the **needs** relationship is a way of expressing polymorphism. As discussed in Section 4.1.3, the two primary approaches to polymorphism supported by programming languages are parametric polymorphism and subtype (inclusion) polymorphism. Languages support parametric polymorphism, parameterized components, through mechanisms such as generic packages in Ada, generic classes in Eiffel, templates in C++, and functors in ML. Subtype polymorphism is supported by inheritance.

The **needs** relationship is a form of what many authors call *bounded polymorphism* [CW85]. If concrete template C **needs** abstract instance A, then the set of acceptable actual parameters for C is bounded by A. A identifies the set of concrete instances that may be used to instantiate C. As several authors have pointed out, bounded polymorphism is difficult to express using subtype polymorphism alone [OW97, LDGM95]. Although it is possible to encode the **needs** relationship using only subtype polymorphism, encoding bounded polymorphism without parametric polymorphism tends to lead to very awkward code. Furthermore, relying on inheritance to achieving bounded polymorphism (as currently must be done in Java) can lead to code bloat and performance penalties for run-time type casting and run-time type checking [BLM96]. Therefore, our clear preference is to use parametric polymorphism, perhaps in conjunction with subtype polymorphism, to encode the **needs** relationship.

Section 5.5 describes how the **needs** relationship is encoded in RESOLVE/Ada95 using parametric polymorphism (Ada generics) and subtype polymorphism (Ada type extension). A similar approach based on C++ templates is used by RESOLVE/C++ [Wei97]. With both Ada and C++, encoding the **needs** relationship is not a trivial matter. Instead of a single language mechanism to express this relationship, such as facility parameters in RESOLVE [SW94]) several language mechanisms must be used in conjunction with each other to achieve the approximate effect. We believe that this points out a significant weakness in current programming languages intended for use in component-based software engineering.

4.6 Chapter Summary

In this chapter we examined how the mechanisms of modern programming languages can be used to encode the component relationships defined in Chapter 3. While the importance of modularity, information hiding, polymorphism, and extendibility has been understood for at least 25 years, the evolution of programming languages has been slow. Only recently has the importance of parametric polymorphism been recognized outside of the academic community and been integrated into widely used languages such as C++. Nevertheless, the most widely used new language, Java, currently does not support parametric polymorphism. Furthermore, most language implementations that do support parametric polymorphism generate independent code for each template instantiation which often leads to code bloat. As we discussed in Section 4.5, parametric polymorphism is very useful for encoding the **needs** relationship.

On balance, the emergence of OOP as a programming paradigm has led to better language support for developing component-based software. Most OOPL's provide good facilities for data abstraction and modularity. Interface-only components, such as abstract classes, are useful for defining structural specification components which may be augmented with behavioral specifications to encode abstract components. While there has been a tendency to use inheritance for many disparate purposes, when used in a disciplined manner, inheritance is a useful tool for encoding relationships such as **implements** and **extends**. Several newer languages, such as Java and Theta, provide distinct mechanisms for expressing specification conformance and implementation inheritance. In the case of Java, use of the keywords `implements` and `extends` for these mechanisms makes encoded relationships easier to identify and understand.

A weakness of most OOPL's is the use of a single mechanism, the class, for both modularization and new type definition. This strategy makes it difficult to define more than one extendable type within a single component. OOPL's typically require breaking encapsulation when one type needs access to another type's representation. Thus, with OOPL's, encoding the **extends** relationship is generally limited to extension of user-defined types.

In order to fully support expression of behavioral relationships between software components, implementation and specification notations need to be better integrated into a single language. While several research languages have taken this approach, to date, such languages have received little attention outside of academia.

CHAPTER 5

BEHAVIORAL RELATIONSHIPS IN RESOLVE/ADA95

In this chapter we demonstrate how the behavioral relationships described in Chapter 3 may be expressed using RESOLVE/Ada95. We begin with an introduction to RESOLVE/Ada95. Then we examine how the language features of Ada are used to express abstract, concrete, and template components, and the behavioral relationships between them. We continue to use the component coupling diagram notation introduced in Chapter 3, to express design time relationships. We also introduce a new component instantiation diagram notation that depicts integration time relationships, i.e., how the components of a particular system have been composed. The chapter concludes with a discussion of the limitations of Ada with respect to its support for expression of behavioral component relationships.

5.1 RESOLVE/Ada95

RESOLVE/Ada95 (RA95) is a discipline for software component engineering that combines the Ada programming language with the specification notation and design discipline of RESOLVE [SW94, WOZ91, Har90]. RESOLVE is three things. First, it is a detailed framework for software component engineering. RESOLVE is also a language which includes two integrated sub-languages: a model-based formal specification language and an imperative, sequential, programming language. Finally, RESOLVE is a discipline with detailed design principles which guide software engineers in the development of high quality software components and systems.

RA95 is a major revision to the RESOLVE/Ada (RA83) discipline described by Hollingsworth [Hol92] and based on the original 1983 Ada language definition [Dep83]. Both RA83 and RA95 apply the component design principles and formal specification notation of RESOLVE to software components implemented in Ada. Unlike RA83, however, RA95 explicitly encodes the behavioral relationships between components using new language mechanisms. RA95 also makes explicit the ACTI view of components as abstract and concrete templates and instances [Edw95, §3.3]. Furthermore, use of new language mechanisms makes most RA95 components much simpler to encode than similar RA83 components.

Ada's strong support for modularity and genericity have long made it a good programming language for implementing parameterized components. The 1995 revision to Ada added substantial new support for object-oriented programming (OOP), a new model of module extension, and more powerful generic parameterization mechanisms. RA95 relies heavily on many of the new Ada language mechanisms. In some aspects, RA95 is similar to the RESOLVE/C++ (RCPP) discipline developed concurrently by Edwards, Weide, and Zhupanov [Wei97]. For example, both RA95 and RCPP rely heavily on inheritance mechanisms to support expression of behavioral relationships. There are many differences, however, between RA95 and RCPP due to differences between Ada and C++ and differences in the basic approach taken.

The RCPP discipline relies heavily on the use of preprocessor macros that serve to make the "source" language of RCPP appear substantially different from that of typical C++. The benefits of this approach include making the RESOLVE and ACTI perspectives more explicit, hiding annoying C++ syntax, and improving maintainability by reducing source redundancy. The approach to RA95 does not require the use of a preprocessor. Implementing components using the RA95 discipline entails coding directly in Ada. One benefit of this approach is that RA95 uses language mechanisms of Ada largely as they were intended to be used. Therefore explaining the rationale for RA95's use of various language mechanisms is easier. Another benefit is that maintenance of RA95 code is maintenance of Ada code. Thus, analysis and maintenance tools available for Ada should be directly applicable to components developed using the RA95 discipline. Finally, a possible practical benefit of this approach is that RA95 may be more accessible to experienced Ada programmers than RCPP is to experienced C++ programmers.

Despite their tremendous complexity, neither Ada nor C++ provides an ideal set (or subset) of language mechanisms for supporting software component engineering. The RESOLVE language, which was designed specifically to support software component engineering, can express notions which are either impossible or extremely awkward to express in Ada or C++. However, one clear advantage Ada and C++ have over RESOLVE is the availability of commercially supported compilers on a wide variety of platforms. Thus, RA95 and RCPP make it easier for software engineers to apply the RESOLVE discipline in the implementation of software components.

The remaining sections of this chapter discuss many, but not all, aspects of RA95. We focus on the expression of behavioral relationships between RA95 components. In doing so, we describe nearly all aspects of RA95 that distinguish it from RA83. We also point out that the RA95 discipline presented in subsequent sections is only one of many possible strategies for applying the RESOLVE discipline to Ada. This version of RA95 attempts to explicitly express as many aspects of the RESOLVE framework as possible. Due to the current immaturity of Ada compilers supporting the 1995 language definition, we have not been able to assess the practicality of this particular approach on large systems (although all components shown in this Chapter do compile

on the current GNAT compiler). As Ada compilers become more robust, the RA95 discipline may be revised to ensure it provides a viable approach for building large systems with available Ada compilers.

5.2 RESOLVE/Ada95 Abstract Components

In RA95, abstract components are either *abstract kernel components* or extensions to other abstract components. This section describes abstract kernel components and, in doing so, most aspects of RA95 abstract components. Section 5.6 describes abstract components that extend other abstract components.

An abstract kernel component is a specification that typically has no dependencies on other specific components. We could use, for example, an abstract kernel component to specify the behavior of a component that provides a queue of integers. In RA95, this specification would be based on a mathematical model of a queue of integers such as a string of integers. Here the phrase "string of integers" refers to the mathematical concept of string defined by string theory and the mathematical notion of integer. The signature of a queue of integers would be characterized by a type name, such as `Integer_Queue`, and the signatures of *primary operations* on that type such as `Enqueue` and `Dequeue`. The functional behavior of the integer queue operations would be described in terms of the queue's mathematical model. For example, the behavior of an initialization operation could be described as returning an empty string of integers. The behavior of an `Enqueue` operation could be described as ensuring that the integer to be enqueued is placed at the right end of the string of integers modeling the queue. While a specification such as this does depend on mathematical string theory and the built-in `Integer` type, it does not depend on any other software components.

An abstract kernel component specifying a queue of integers would be an *abstract instance*. A more useful specification would be an *abstract template* describing the behavior of a generic queue. Such an abstract template would be parameterized by the type of item to be contained in instances of the template. An abstract instance of a queue abstract template might depend on another specific component providing the item type. The queue abstract template itself, however, need not depend on any other specific component. In order to support greater reusability, most RA95 abstract kernel components are abstract templates as opposed to abstract instances.

One of the aspects of the RESOLVE framework that most distinguishes it from other disciplines is its use of the *swapping* paradigm. RESOLVE uses swapping instead of assignment as the primary method for data movement. The use of swapping provides profound benefits and is one of the cornerstones of the RESOLVE framework[HW91]. The `Swap` operation simply exchanges the values (both the abstract and concrete values) of its two operands. `Swap` provides the efficiency of assignment implemented by copying a pointer (shallow copying) while maintaining value

semantics like assignment implemented by copying a value (deep copying). Using swapping avoids the aliasing problems which result from shallow copying and which significantly complicate formal justification of the **implements** relationship between implementations and specifications. An important insight on which the RESOLVE framework capitalizes is that deep copying is rarely needed when components are designed to use swapping. RA95's support for swapping as the primary data movement operation has significant implications that affect the way in which Ada is used to encode RA95 abstract kernel components.

An RA95 abstract kernel component is encoded as an Ada package, usually generic, which exports an *abstract type* and *abstract primitive operations* of the type. The exported abstract type is declared as an *abstract tagged limited private type*. In Ada abstract types are *tagged types* from which no objects (variables) may be declared. Ada tagged types are types from which new user-defined types may be derived using *type extension*, a form of inheritance. Primitive operations of a tagged type are those operations declared in the same package as the tagged type and having at least one parameter of the tagged type. In Ada, only primitive operations are inherited by new types derived from another tagged type. (In OOP terminology, the exported tagged type is a *class* and the primitive operations are its *member functions*.)

The type exported by a kernel abstract component is limited private which means that assignment and equality operations are not automatically defined for objects of the type. Since RA95 uses swap instead of assignment as the primary data movement operation, a primitive assignment operation is unnecessary. In situations where the deep copying functionality of assignment is needed, it may be added with an extension component as described in Section 5.6. While it would be technically possible to support deep copy assignment in all kernel components, the automatically defined equality is an Ada *function* and is not compatible with RA95 as explained below. Thus, RA95 uses Ada's limited private types to prevent automatically defined assignment and equality.

An unfortunate requirement of Ada functions is that their parameters must be *"in" mode*. The intent of this requirement is to ensure that a function implementation does not change the values of the actual parameters used in a function call. The problem with this requirement is that it is both overly restrictive and largely ineffective in achieving its intended purpose. For example, consider an operation to determine if two queues are equivalent, i.e., whether they represent the same abstract value. An implementation of this operation should be permitted to remove and compare items from each queue as long as it returns the queues back to their original abstract values before completing execution. Ada, however, does not allow an "in" mode formal parameter to be passed as an actual parameter to another operation where an *"in out" mode* parameter is required. Therefore, operations of the encapsulated representation type cannot be used to disassemble the queues in order to compare individual queue items. A method of circumventing this limitation would be to represent a component's

data structure as a pointer (an Ada *access* type) to an unencapsulated data structure. This is clearly unacceptable as a general solution since it precludes constructing data representations from other encapsulated types. Thus, in general, RA95 prohibits the use of Ada functions since they preclude implementations that need to directly or indirectly alter formal parameter values[25].

In the special case of the equality function, a more extreme general solution is possible. We could require that all types support equality and thus make it possible to provide encapsulation with *non-limited* private types. Then an implementation of an equality operation could call the equality operation(s) of its constituent representation type(s). This approach is unacceptable from the RESOLVE perspective since for all but simple types equality testing tends to be very expensive and is often unnecessary (and theoretically uncomputable for some types). Therefore, RA95 kernel components only define an equality operation (with an Ada *procedure*) on types for which equality is an essential operation. Like copying operations, RA95 equality testing operations may be provided as extensions.

The RESOLVE counterpart to an "in" mode parameter is a *preserves mode* parameter. An operation is permitted to change the *representation value* of a preserves mode parameter as long as there is no net change in the *abstract value* of the parameter. In addition to preserves mode, RESOLVE parameter modes include *alters mode*, *produces mode*, and *consumes mode*. An operation may change the abstract value of an alters mode parameter. The initial abstract value of a produces mode parameter is irrelevant to an operation's effect. The initial value of a consumes mode parameter generally is relevant to an operation's effect and its final returned value must be an initial value of its type. In RA95, operations are encoded as procedures. All parameters, regardless of the RESOLVE mode, use Ada's "in out" mode except for preserves mode parameters of built-in scalar types. In this situation "in" mode is used in order to allow scalar literal values as actual parameters. Formal comments identify the RESOLVE mode of each procedure parameter in RA95. While RESOLVE's parameter modes convey design intent, their primary purpose is to simplify specification of operation preconditions and postconditions.

All RESOLVE components that define types provide the three *standard operations*: `Initialize`, `Finalize`, and `Swap`. `Initialize` sets the abstract value of an object to an initial value of its type. RESOLVE guarantees that all objects are automatically initialized when they come into existence. `Finalize` is typically used to reclaim system resources allocated to an object. RESOLVE guarantees that all objects are automatically finalized immediately before they cease to exist. The functional behavior of `Finalize` usually does not need to be specified. `Swap` exchanges the abstract values of two objects as described above.

[25]RESOLVE/C++ does not suffer from this annoyance since C++ function parameters may be modified.

RA95 uses Ada's *controlled types* to achieve automatic initialization and finalization of all objects except built-in scalars. A controlled type is a tagged type derived from either `Controlled` or `Limited_Controlled`, two types defined in the built-in package `Ada.Finalization`. `Limited_Controlled` has two primitive operations: `Initialize` and `Finalize`. These operations have null-body implementations which may be overridden as necessary for descendent types derived from `Limited_Controlled`. While the exact details are quite involved, Ada basically assures that an `Initialize` operation is automatically called when an object of a controlled type comes into existence. Similarly, Ada assures that a `Finalize` operation is automatically called immediately before a controlled type object ceases to exist.

Abstract kernel components derive their exported type from `Limited_Controlled` with a *null record extension* in the *private* part of the package. The private derivation assures that `Initialize` and `Finalize` cannot be called explicitly by clients. The null record extension adds no data representation to the type Limited_Controlled and thus effectively leaves the data representation of the exported type unspecified. (Due to Ada accessibility rules, the derivation from `Limited_Controlled` must take place in the abstract kernel component and cannot be placed in a concrete component, which would be a more appropriate location.) In order to derive the exported type from `Limited_Controlled`, the package encoding a kernel abstract component **uses** the built-in package `Ada.Finalization`. The `with` *context clause* in the *global context section* of RA95 components expresses this **uses** relationship. In RESOLVE, the global context section lists all non-parametric dependencies. We discuss the **uses** relationship in RA95 in Section 5.3.

In addition to the implicit operations `Initialize` and `Finalize` inherited from `Limited_Controlled`, all abstract kernel components explicitly export `Swap` and other *abstract operations*. An Ada abstract operation is a primitive operation that must be overridden with an implementation for a non-abstract type derived from the abstract type. The exported abstract type and abstract operations along with formal specifications embedded in structured comments provide the kernel abstract component's signature and behavioral specification.

Abstract kernel components are typically abstract templates parameterized by other components. These abstract templates are encoded in RA95 as generic packages. The abstract template parameters are included in a *specification parameters section* and are encoded as Ada generic formal parameters. Each component imported as a specification parameter is encoded as a *limited private generic formal type parameter*. Each generic formal type parameter must have an associated *generic formal subprogram parameter* importing the `Swap` operation for that type. Other generic formal subprograms may be used to place constraints on imported components. Ada's subprogram default box specification (<>) is used with all generic formal subprograms to simplify generic package instantiation. We discuss specification parameters further in Section 5.5.

The name of a kernel abstract component is the name of the abstraction being described, prefixed with `AT_` or `AI_`, depending on whether it is an abstract template or an abstract instance. For example, `AT_Queue` is the package name of a kernel abstract component describing a queue. The name of the exported type is usually the name of the abstraction being described. For example, the type exported from `AT_Queue` would be named `Queue`. In some cases, the type name may be shortened to improve the readability of the code. Note that an abstract template only requires a package specification and no associated package body.

Figures 5.1 and 5.2 show the RA95 code for the `AT_Queue` component. The only aspect of this code not described above is the design choice of which operations should be primary operations included in the kernel component. The choice of primary operations for `AT_Queue` reflects the RESOLVE design principle that a kernel component should be as simple as possible while providing a client controllability and observability over the component's abstract state [WEH+96]. While other queue operations will inevitably be useful in various contexts, these *secondary operations* may be added as extensions and implemented by layering on top of the primary operations provided by the kernel component.

5.3 The RESOLVE/Ada95 uses Relationship

As we have mentioned before, Ada's `with` clause is the primary mechanism for encoding the **uses** relationship between two Ada packages. However, since one of the goals of the RESOLVE discipline is to minimize fixed design dependencies, the **uses** relationship is employed quite sparingly in the design of RA95 components. In order to minimize coupling and maximize reusability, the RESOLVE discipline suggests that components (abstract and concrete) be *fully parameterized* [SW94, pp. 34,40]. That is, all dependencies which can be deferred should be deferred rather than fixed. In terms of software component relationships, this means that, whenever possible, components should be designed with dependencies expressed in terms of the **needs** relationship rather than the **uses** relationship. Usually, a **uses** relationship between two RA95 components either is associated with a **needs**, **implements**, or **extends** relationship (as discussed in subsequent sections), or is a dependency on a component in Ada's predefined language environment (a built-in package).

The abstract template `AT_Queue` shown in Figures 5.1 and 5.2 **uses** two Ada built-in packages. The `with` clause in the global context section explicitly documents a fixed dependency on the package `Ada.Finalization` as discussed in Section 5.2. There is also an implicit dependency on the built-in package `Standard` which defines all of Ada's pre-defined identifiers, primarily those of the built-in types and operations. In this case, `AT_Queue` requires visibility to the `Standard` package for the definition of the type `Integer` referenced in the declaration of the `Get_Length` procedure. All Ada packages have implicit visibility to `Standard`. No "`with Standard`" clause is

```
------------------------------------------------------------------------
--
-- Component: AT_Queue
-- Relations: -
--  Comments: -
--
------------------------------------------------------------------------
-- Global Context ------------------------------------------------------
------------------------------------------------------------------------

with Ada.Finalization;

------------------------------------------------------------------------

generic

    ------------------------------------------------------------------
    -- Specification Parameters --------------------------------------
    ------------------------------------------------------------------

    type Item is limited private;
    with procedure Swap (left, right : in out Item) is <>;

    ------------------------------------------------------------------

package AT_Queue is

    ------------------------------------------------------------------
    -- Interface -----------------------------------------------------
    ------------------------------------------------------------------

    type Queue is abstract tagged limited private;

    --! type Queue is modeled by string of Item
    --!     exemplar q
    --!     initialization ensures
    --!         q = empty_string

    ------------------------------------------------------------------

    procedure Enqueue (
            q : in out Queue;               --! alters q
            x : in out Item                 --! consumes x
        ) is abstract;

    --! ensures
    --!     q = #q * <#x>
```

Figure 5.1: Package Specification for AT_Queue

```
    ------------------------------------------------------------------

    procedure Dequeue (
            q : in out Queue;                --! alters q
            x : in out Item                  --! produces x
        ) is abstract;

    --! requires
    --!     q /= empty_string
    --! ensures
    --!     #q = <x> * q

    ------------------------------------------------------------------

    procedure Get_Length (
            q      : in out Queue;           --! preserves q
            length : in out Integer          --! produces length
        ) is abstract;

    --! ensures
    --!     length = |q|

    ------------------------------------------------------------------

    procedure Swap (
            left  : in out Queue;            --! alters left
            right : in out Queue             --! alters right
        ) is abstract;

    --! ensures
    --!     left = #right and right = #left

    ------------------------------------------------------------------

private

    type Queue is abstract new Ada.Finalization.Limited_Controlled
        with null record;

    ------------------------------------------------------------------

end AT_Queue;
```

Figure 5.2: Package Specification for AT_Queue (Continued)

necessary. In CCD's, we do not depict **uses** relationships involving the `Standard` and `Ada.Finalization` packages.

Unlike RESOLVE, Ada does not provide a swap operation for built-in types. In order for a built-in type to serve as an actual package parameter in component composition, the `Swap` procedure must be available. The special RA95 package `CI_Scalar_Operations` provides `Swap` procedures for the Ada built-in types: `Boolean`, `Integer`, `Character`, and `Float`. This package also provides (in procedure form) commonly used operations for copying, equality testing, and order testing. As discussed in Section 5.2, Ada functions with in-mode only parameters present serious problems for component composition in RA95. We discuss a related issue, automatic initialization of built-in scalars, in Section 5.9.

5.4 The RESOLVE/Ada95 implements Relationship

In RA95, concrete components are either *concrete kernel components* or implementations of abstract extension components. This section describes concrete kernel components and expression of the **implements** relationship between concrete and abstract kernel components. Section 5.6 describes concrete components that implement abstract extension components.

RA95 concrete kernel components, implementations of abstract kernel components, are encoded in Ada as *generic child unit packages*. A concrete kernel component is a generic child unit of the abstract kernel component that it implements. As discussed in Sections 3.3 and 4.3, a single concrete component may implement more than one abstract component. In RA95, however, each implementation is permanently linked to a single specification that it implements. Thus, an **implements** relationship is encoded directly into each concrete component in RA95. Limiting each concrete template to a single **implements** relationship sacrifices little in practice and results in simpler RA95 code.

Child units are a part of the *hierarchical library* structure added to Ada in the 1995 language definition. A child unit is an Ada package that has full visibility to the package specification of another unit, its parent unit. Since a child unit may be the parent unit to other child units, a hierarchy of related library units may be constructed. Thus, a child unit **uses** its parent unit as well as any other units above it in the hierarchy (e.g., its parent unit's parent unit). The association of a child unit to its parent is encoded by the package name of the child unit. A child unit package name has as a prefix its parent unit's package name followed by a period. Thus, the package `AT_Queue.CT_2` is a child unit of the package `AT_Queue`. As a child unit, `AT_Queue.CT_2` has implicit visibility to all of the `AT_Queue` package specification, including generic formal parameters and the private part. A child unit does not need a `with` clause for visibility to its parent unit.

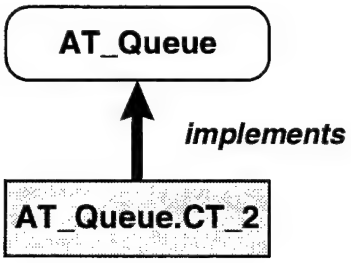


Figure 5.3: The **implements** Relationship in RA95

RA95 concrete component names encode (the claim of) an **implements** relationship. The package name `AT_Queue.CT_2`, for example, identifies the "second" concrete template that **implements** the queue abstract template. (The name is best interpreted from right to left.) By RA95 convention, numbers are used to distinguish concrete templates which serve as multiple implementations of the same abstract component. Use of this convention helps avoid long and possibly misleading component names. Figure 5.3 shows the component coupling diagram depicting the **implements** relationship between concrete template `AT_Queue.CT_2` and abstract template `AT_Queue`.

Recall from Chapter 3 that a CCD depicts design dependencies that are independent of any particular use of the components involved. In contrast, a *component instantiation diagram* (CID) shows how the components in a particular system are coupled with each other. Figure 5.4 shows the RA95-specific component instantiation diagram depicting the parent-child relationship between abstract and concrete kernel components. Note that this parent-child relationship is not a component relationship; it is merely one possible technique useful for expressing the **implements** relationship in Ada. In this example, the abstract parent unit `AT_Queue` is enclosed in the concrete child unit `AT_Queue.CT_2`. This notation conveys the idea that the definition of the child unit "includes" that of the parent unit.

CID's, like CCD's, depict abstract components with clear rounded boxes and concrete components with shaded rectangular boxes. Specification parameters are shown as arrowheads along the top of abstract components. In Figure 5.4, `Item` is the only specification parameter shown. The generic subprogram parameter for `Swap` for type `Item` is implied since all RESOLVE types provide `Swap`. The type exported from a component is shown as an arrowhead on the right side of the box. Implementation parameters (discussed later in this section) are shown as arrowheads along the bottom of concrete components. A more detailed version of this notation

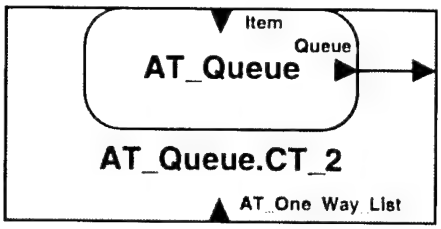


Figure 5.4: Concrete Child Coupled To Abstract Parent

(not shown) includes the names of each exported operation in the lower right corner of the box. CID's in subsequent sections will build upon this diagram.

An RA95 concrete kernel component exports a concrete type and concrete primitive operations of that type. The concrete type exported is a *private type extension* of the abstract type exported by its parent unit. The type extension forms an inheritance link between the concrete and abstract types. This link is depicted in Figure 5.4 as the line connecting the abstract type exported from `AT_Queue` (`AT_Queue.Queue`) to the concrete type exported from `AT_Queue.CT_2` (`AT_Queue.CT_2.Queue`). This inheritance link requires the child unit implementation to provide concrete primitive operations conforming (structurally) to the abstract type's abstract primitive operations. Therefore, an Ada compiler will ensure that a concrete component at least provides the structural interface described by the abstract component that it implements. Of course, Ada compilers provide no help in checking whether a concrete component actually provides the behavior specified by an abstract component. The formal semantics, proof rules, and specification sub-language of the RESOLVE language do provide support for formally verifying the **implements** relationship, though.

Unlike abstract components, concrete components require both a package specification and a package body. The package specification includes a public *interface section* in which the exported type is derived from the abstract type exported by the parent unit. Placing this type derivation in the public section provides a *partial view* of the exported type. The partial view provides a client of the implementation visibility to the exported concrete operations, but not to the type's private data representation. The interface section also includes a concrete subprogram header corresponding to each abstract operation in the parent unit. The package body includes all of the corresponding subprogram bodies. The package body also may include a *local operations* section for local (private) subprograms used internally to implement the exported operations.

The private part of the package specification includes a *representation section* and an *implicit operations section.* The representation section defines the data structure used to maintain state information for each object declared from the concrete component. The *full view* of the exported type is a single field *record extension* of the abstract type in the parent unit. To ensure composability of components, the single-record extension field is always named `rep`. The type of the `rep` field is the type of the component's data representation. If the representation has more than one constituent component, the components are encapsulated in a single RA95 `Recordn` component which serves as the type of `rep`. `Recordn` components are special RA95 components that export `Swap` and field selection operations. `Recordn` components compose with other RA95 components and are responsible for encapsulating various memory management strategies. Formal comments in the representation section describe the implementation's convention and correspondence. The convention specifies any representation invariants and the correspondence describes the abstraction relation, a mapping between the representation states and the model states.

The implicit operations section is where the inherited `Initialize` and `Finalize` operations may be overridden, if necessary. The exported concrete type inherits concrete null-bodied implementations of these operations from its abstract parent (which inherits them from `Limited_Controlled`). For typical components, the inherited `Initialize` and `Finalize` provide the correct behavior. However, some components will need to explicitly override these operations, especially `Initialize`.

Since a concrete kernel component is a child unit of an abstract kernel component, it has direct visibility to any specification parameters. Thus, the specification parameters are implicit parameters to the concrete kernel component. This idea is conveyed in Figure 5.4 since the specification parameter `Item` on the top edge of `AT_Queue` is also on the top edge of `AT_Queue.CT_2`. In addition to implicit specification parameters, a concrete component also may have additional parameters which, if present, are included in the *implementation parameters section.* Implementation parameters are encoded in RA95 as *generic formal packages*, types, and subprograms for which actual parameters must be supplied through instantiation. We discuss how implementation parameters are used to encode the **needs** relationship in Section 5.5.

Figures 5.5 and 5.6 show the package specification for the concrete template `AT_Queue.CT_2` which **implements** `AT_Queue` shown in Figure 5.1. Figures 5.7 and 5.8 show the package body for `AT_Queue.CT_2`. These may be compared with the similar concrete template implementing a stack shown in Figure 3.11. We discuss the **needs** relationship encoded by this component in the next section.

5.5 The RESOLVE/Ada95 needs Relationship

As we discussed in Section 5.3, a fixed dependency on an Ada program unit is encoded in RA95 by a `with` clause in the global context section. For example,

```
------------------------------------------------------------------------
--
-- Component: AT_Queue.CT_2
-- Relations: implements AT_Queue, needs AT_One_Way_List
--  Comments: queue implemented with a One_Way_List representation
--
------------------------------------------------------------------------
-- Global Context ------------------------------------------------------
------------------------------------------------------------------------

with AT_One_Way_List;

------------------------------------------------------------------------

generic

    ------------------------------------------------------------------
    -- Implementation Parameters -----------------------------------
    ------------------------------------------------------------------

    with package AI_One_Way_List is new AT_One_Way_List (Item => Item);

    type List is new AI_One_Way_List.List with private;

    ------------------------------------------------------------------

package AT_Queue.CT_2 is

    ------------------------------------------------------------------
    -- Interface ---------------------------------------------------
    ------------------------------------------------------------------

    type Queue is new AT_Queue.Queue with private;

    ------------------------------------------------------------------

    procedure Enqueue (
            q : in out Queue;
            x : in out Item
        );

    ------------------------------------------------------------------

    procedure Dequeue (
            q : in out Queue;
            x : in out Item
        );
```

Figure 5.5: Package Specification for AT_Queue.CT_2

```
    ------------------------------------------------------------------

    procedure Get_Length (
            q      : in out Queue;
            length : in out Integer
        );

    ------------------------------------------------------------------

    procedure Swap (
            left  : in out Queue;
            right : in out Queue
        );

    ------------------------------------------------------------------

private

    ------------------------------------------------------------------
    -- Representation ------------------------------------------------
    ------------------------------------------------------------------

    type Queue is new AT_Queue.Queue with
        record
            rep : List;
        end record;

    --! convention
    --!     true
    --! correspondence
    --!     q = q.rep.left * q.rep.right

    ------------------------------------------------------------------
    -- Implicit Operations -------------------------------------------
    ------------------------------------------------------------------

    -- Inherited null-bodied Initialize and Finalize

    ------------------------------------------------------------------

end AT_Queue.CT_2;
```

Figure 5.6: Package Specification for AT_Queue.CT_2 (Continued)

```
------------------------------------------------------------------------
--
-- Component: AT_Queue.CT_2
-- Relations: implements AT_Queue, needs AT_One_Way_List
--  Comments: queue implemented with a One_Way_List representation
--
------------------------------------------------------------------------
-- Global Context ------------------------------------------------------
------------------------------------------------------------------------

------------------------------------------------------------------------

package body AT_Queue.CT_2 is

    ------------------------------------------------------------------------
    -- Implicit Operations -------------------------------------------------
    ------------------------------------------------------------------------

    -- Inherited null-bodied Initialize and Finalize

    ------------------------------------------------------------------------
    -- Interface Operations ------------------------------------------------
    ------------------------------------------------------------------------

   procedure Enqueue (
            q : in out Queue;
            x : in out Item
        ) is
    begin
        Move_To_Finish (q.rep);
        Add_Right (q.rep, x);
    end Enqueue;

    ------------------------------------------------------------------------

    procedure Dequeue (
            q : in out Queue;
            x : in out Item
        ) is
    begin
        Move_To_Start (q.rep);
        Remove_Right (q.rep, x);
    end Dequeue;
```

Figure 5.7: Package Body for AT_Queue.CT_2

```
    ------------------------------------------------------------------------

    procedure Get_Length (
            q      : in out Queue;
            length : in out Integer
        ) is
    begin
        Move_To_Start (q.rep);
        Get_Right_Length (q.rep, length);
    end Get_Length;

    ------------------------------------------------------------------------

    procedure Swap (
           left  : in out Queue;
           right : in out Queue
      ) is
    begin
        Swap (left.rep, right.rep);
    end Swap;

    ------------------------------------------------------------------------

end AT_Queue.CT_2;
```

Figure 5.8: Package Body for AT_Queue.CT_2 (Continued)

dependencies on standard Ada library units such as `Text_IO` (which defines standard input and output routines) must be encoded as fixed dependencies. Even so, many of Ada's standard library units are generic units or units exporting abstract types (such as `Ada.Finalization` used by `AT_Queue` in Figure 5.1) and thus are structural specifications of many possible implementations.

The **needs** relationship expresses a deferred dependency – dependence on a specification of behavior rather than on a specific implementation of behavior. In RA95, as in RESOLVE, the **needs** relationship is encoded using parameterized components. However, RA95 uses Ada's type extension (inheritance) to constrain concrete components serving as actual parameters to be implementations of the appropriate abstract component. This approach was not possible in RA83 which used (potentially long) lists of generic formal subprogram parameters to constrain type parameters [Hol92]. Nevertheless, encoding the **needs** relationship in RA95 is somewhat complex as it makes use of a combination of several of Ada's more advanced language mechanisms.

The **uses** relationship between a concrete template and an abstract component is encoded in RA95 using a pair of related generic formal parameters and a `with` clause in the global context section. The `with` clause names the abstract component upon

which the implementation depends. The first generic parameter is a *generic formal package parameter* and the second is a *generic formal derived type*. Both of these generic parameters are new language mechanisms added to Ada in the 1995 language definition. The actual parameter corresponding to the formal package parameter must be an instance of the abstract package component being used. The actual parameter corresponding to the formal derived type must be a tagged type derived from the abstract type exported by the first actual parameter. Together, these two parameters ensure that the imported type has all of the primitive operations specified in the abstract package. Therefore, the imported concrete type is constrained to have been exported from a concrete component which **implements** the abstract component being used. Of course, only the structural aspects of the **implements** relationship are checked by the compiler.

Figure 5.9 shows a CCD depicting the **uses** relationship between the concrete template `AT_Queue.CT_2` and the abstract template `AT_One_Way_List`. The `CT_2` implementation of `AT_Queue` uses a `One_Way_List` as its representation as shown in Figures 5.5-5.7. `One_Way_List` is a list abstraction that supports list traversal in one direction (thus allowing singly-linked list implementations). It is modeled by a pair of strings which conceptually represent the left and right parts of the list. Traversing the list from left to right is modeled by moving the leftmost item in the right string to the rightmost item in the left string. Thus, the current position within a `One_Way_List` may be viewed as just to the left of the leftmost item in the right string.

`AT_One_Way_List` specifies the following operations:

- `Advance` moves the current position one item to the right.
- `Move_To_Finish` moves the current position to the right of the rightmost item.
- `Move_To_Start` moves the current position to the left of the leftmost item.
- `Add_Right` inserts an item at the left end of the right string.
- `Remove_Right` removes the item at the left end of the right string.
- `Get_Left_Length` returns the length of the left string, and
- `Get_Right_Length` returns the length of the right string.

Figures 5.5-5.8 demonstrate how the **needs** relationship depicted in Figure 5.9 is encoded in RA95. The `with` clause in the global context section (Figure 5.5) expresses the fixed dependency of `AT_Queue.CT_2` on the specification `AT_One_Way_List`. The first generic parameter in the implementation parameters section is the formal package `AI_One_Way_List`. The actual parameter corresponding to this formal must be a package which is an instance of `AT_One_Way_List`. Furthermore, this instance must have been instantiated with the same `Item` type as the instance of `AT_Queue` which

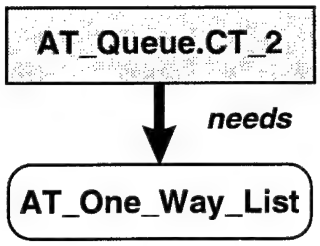


Figure 5.9: The **needs** Relationship in RA95

serves as the parent of an instance of `AT_Queue.CT_2`. References to `AT_Queue` within `AT_Queue.CT_2` refer to the *instantiation* of `AT_Queue` that serves as the parent unit of the instantiation of `AT_Queue.CT_2`.

The second generic parameter is the formal derived type `List`. The actual parameter corresponding to `List` must be a concrete type derived (possibly indirectly) from the abstract `List` type exported by the package serving as the actual parameter to `AI_One_Way_List`. Note that in a component instantiation diagram, a single implementation parameter corresponds to the pair of generic parameters required to express the **needs** relationship. For example, in Figure 5.4, the single implementation parameter `AT_One_Way_List` expresses the **needs** relationship just described.

At the programming language level, this encoding of **needs** assures that the imported `List` type comes with implementations for all of the operations specified in `AT_One_Way_List`. In terms of component relationships, this strategy encodes `AT_Queue.CT_2`'s dependency on the behavior specified by `AT_One_Way_List` without making `AT_Queue.CT_2` dependent on a specific implementation of `AT_One_Way_List`. In Sections 5.7.1 and 5.8 we present examples of how RA95 components may be composed through instantiation of concrete templates.

5.6 The RESOLVE/Ada95 extends Relationship

In this section, we first discuss how the **extends** relationship is encoded between two abstract components in RA95. Then, we discuss how a layered implementation of an abstract extension component is encoded. We present example RA95 code for both types of components.

5.6.1 Abstract Extension Components

The RESOLVE discipline encourages careful design of kernel components with a minimally sufficient set of primary operations. New functionality is then added —

usually one operation at a time — with components which extend the behavior of existing components. The ideal choice of primary operations for a kernel component is seldom obvious. One possible approach is to design kernel components with no operations (just an exported type), and then to add all operations as extensions. This approach is being used with RESOLVE/C++ [Wei97]. The RESOLVE/Ada95 discipline currently takes the more conventional approach of including primary operations with the kernel component, as described in Section 5.2.

The **extends** relationship describes the behavioral relationship between two abstract components as discussed in Section 3.5.1. Encoding the **extends** relationship using the language mechanisms of Ada presents some challenges. Difficulties arise primarily from needing to rely on Ada's single inheritance mechanism, type extension. One of the difficulties encountered when attempting to encode component relationships in Ada involves the expression of multiple relationships for a single component. Ada does not directly support multiple inheritance, which has proven useful in RESOLVE/C++, and does not distinguish between specification inheritance (structural interface conformance) and implementation inheritance, as do newer OOPL's, such as Java.

RA95 uses *mixin inheritance* to express multiple dependencies [Int95a, §4.6.2]. Mixin inheritance is expressed in Ada by deriving an exported type from another type imported as a generic parameter. In order to apply multiple extensions to a component, use of mixin inheritance requires chaining extensions together to form a linear inheritance path. To support structural interface conformance of each concrete extension component to an abstract extension component requires an inheritance chain that alternates between abstract and concrete types.

An abstract extension component is encoded in RA95 as a generic child unit of the abstract component it extends. By convention, the package name of the child unit is the name of the parent unit followed by a period, followed by the string "`With_`", followed by a name describing the new functionality. For example, the abstract template `AT_Queue.With_Reverse` extends `AT_Queue` by a specification of the queue reverse functionality. In general, it might be possible for a single abstract extension component to extend more than one abstract component. In RA95, however, each abstract extension is permanently linked to a single specification that it extends. Thus, the **extends** relationship is encoded directly into each abstract extension component in RA95.

Figure 5.10 shows the RA95-specific CID depicting the parent-child relationship between a kernel abstract component and an abstract extension component. In this example, the abstract parent unit `AT_Queue` is enclosed in `AT_Queue.With_Reverse`, the abstract child unit. Figure 5.10 appears similar to Figure 5.4 since RA95 uses similar language mechanisms, type extension and hierarchical library units, to encode **implements** and **extends**. The primary difference is that `AT_Queue.With_Reverse`

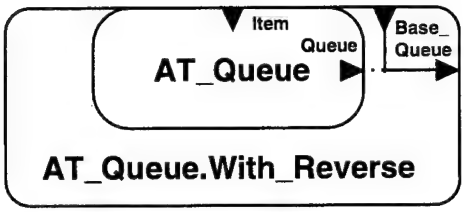


Figure 5.10: Abstract Parent and Abstract Child Extension

exports an abstract type whereas `AT_Queue.CT_2` exports a fully-implemented concrete type.

The specification parameters section of an abstract extension component has a single specification parameter which serves to support chaining of extensions. This parameter is encoded as a formal generic derived type. The name of the formal parameter is the name of the exported type prefixed with the string "`Base_`". The actual parameter corresponding to this formal parameter is constrained to be a concrete type derived from the abstract type exported by the abstract parent unit being extended. For example, `AT_Queue.With_Reverse`, shown in Figure 5.10, has the specification parameter `Base_Queue`. The dotted line between the type exported from `AT_Queue` and the line extending from the specification parameter conveys the constraint on the actual parameter for `Base_Queue`. Note that the actual parameter need not be derived directly from the abstract type exported by (an instance of) the parent unit. The actual parameter may be the result of many extensions to that type. This flexibility allows multiple extensions of the same abstract component to be chained together.

The interface section of an abstract extension component includes an exported abstract type publicly derived from the formal generic type parameter. The name of the exported type is the same as the name of the abstract type from which it is derived. (The exported type's full name includes its package name, thus distinguishing it from its parent type.) This derivation provides the mixin inheritance. Furthermore, the exported type is a null record extension of the type imported by the generic parameter. The null record extension means that the abstract extension component does not augment the representation of the imported type. In Figure 5.10, the line connecting the `Base_Queue` specification parameter to the abstract type exported by `AT_Queue.With_Reverse` depicts the mixin inheritance link. The abstract type exported by an instance of `AT_Queue.With_Reverse` has as primitive operations the abstract queue reverse operation specified in the extension *plus* all concrete operations of the type used as the actual parameter for `Base_Queue`.

The remainder of the interface section includes usually one, but possibly more, abstract subprogram specifications. Each subprogram specification corresponds to

an operation providing new functionality to the abstract component being extended. Structured comments formally describe the functional behavior of the abstract operations.

Figure 5.11 shows the Ada package specification encoding the abstract template `AT_Queue.With_Reverse`. No package body is required as with abstract kernel components. Note that normally the added operation to reverse a queue would be named `Reverse` instead of `Reverse_Queue`. Since `reverse` is a reserved word in Ada, however, it cannot be used as an operation name. Also note that `reverse` is a built-in operation of the string theory mathematics in RESOLVE's specification sub-language. This explains how the ensures clause of `Reverse_Queue` may be expressed so concisely.

5.6.2 Implementation of Abstract Extension Components

Recall from Section 3.5.2 the three approaches to implementing an extension component: layered, direct, and coupled implementations. Since the RESOLVE discipline primarily advocates use of the layered approach, the strategy for encoding the **extends** relationship in RA95 was designed to best support layered implementations. In this section we discuss how to encode a layered implementation of an abstract extension component.

The layered implementation of a concrete extension component is encoded in RA95 as a generic child unit of the extension component it implements. The prefix of child unit's package name is the name of the abstract extension component (the parent unit name) followed by a period. Just like implementations of kernel components, the name ends with the string "`CT_`" followed by a number identifying the specific implementation. For example, the concrete extension component name `AT_Queue.With_Reverse.CT_1` denotes the first concrete template that implements the reverse component that extends the queue abstract template. Again, the component name is best interpreted from right to left.

Figure 5.12 shows the RA95-specific CID depicting the parent-child relationships between a kernel abstract component, an abstract extension component, and a layered concrete extension component. In this example, the abstract extension component `AT_Queue.With_Reverse` (Figure 5.11) is enclosed in `AT_Queue.With_Reverse.CT_1`, the concrete child unit.

A concrete extension component consists of a package specification and body which are similar in structure to those of a kernel concrete component. The package specification includes an interface section, a representation extension section, and possibly an implementation parameters section. The interface section includes a concrete exported type and concrete subprogram specifications. Just as with kernel concrete components, the exported type is a private type extension of the abstract type exported by its parent unit. This inheritance link is shown in Figure 5.12 as the line connecting the abstract type exported from `AT_Queue.With_Reverse` to the concrete

```
------------------------------------------------------------------------
--
-- Component: AT_Queue.With_Reverse
-- Relations: extends AT_Queue
--  Comments: 'Reverse' is an Ada reserved word, hence Reverse_Queue
--
------------------------------------------------------------------------
-- Global Context ------------------------------------------------------
------------------------------------------------------------------------

------------------------------------------------------------------------

generic

    ------------------------------------------------------------------
    -- Specification Parameters ------------------------------------
    ------------------------------------------------------------------

    type Base_Queue is new AT_Queue.Queue with private;

    ------------------------------------------------------------------

package AT_Queue.With_Reverse is

    ------------------------------------------------------------------
    -- Interface ---------------------------------------------------
    ------------------------------------------------------------------

    type Queue is abstract new Base_Queue with null record;

    ------------------------------------------------------------------
    -- Added Operations --------------------------------------------
    ------------------------------------------------------------------

    procedure Reverse_Queue (
            q : in out Queue                --! preserves q
        ) is abstract;

    --! ensures
    --!     q = reverse(#q)

    ------------------------------------------------------------------

end AT_Queue.With_Reverse;
```

Figure 5.11: Package Specification for AT_Queue.With_Reverse

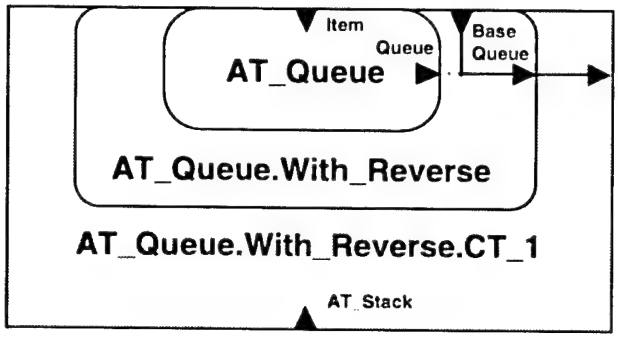


Figure 5.12: Abstract Parent and Concrete Child Extension

type exported by `AT_Queue.With_Reverse.CT_1`. In this example, both types are named `Queue` (as is the type exported by `AT_Queue`).

Derivation of the exported concrete type in the public part of the package provides a partial view of the type and ensures that the concrete operations imported by the parent unit are re-exported along with the concrete exported type. For example, the actual parameter for `Base_Queue` in Figure 5.12 may include many extensions to `AT_Queue`. However, the body of `AT_Queue.With_Reverse.CT_1` only has visibility to the operations described by `AT_Queue.With_Reverse` (which include those of `AT_Queue`).

The interface section also includes one concrete subprogram specification corresponding to each abstract operation added by the parent unit to describe new functionality. The representation extension section is in the private part of the package and contains the declaration of the exported type providing its full view. The exported type is a null record extension of the abstract type exported by the parent unit. The implementation parameters section, if present, serves the same role as in kernel concrete components. Figure 5.12 shows the implementation parameter `AT_Stack` used to express the **needs** relationship `AT_Stack`. Thus, `AT_Queue.With_Reverse.CT_1` is layered on top of implementations of both `AT_Stack` and `AT_Queue`.

The package body of a concrete extension component includes one subprogram body corresponding to each added operation. Like its kernel component counterpart, it also may include a local operations section to provide local subprograms used to implement the exported operations.

Figure 5.13 shows the Ada package specification for the concrete extension component `AT_Queue.With_Reverse.CT_1`. Figure 5.14 shows the corresponding package body with the layered implementation of the `Reverse_Queue` operation.

```
------------------------------------------------------------------------
--
-- Component: AT_Queue.With_Reverse.CT_1
-- Relations: implements AT_Queue.With_Reverse, needs AT_Stack
--  Comments: Reverse_Queue reverses a queue using a stack
--
------------------------------------------------------------------------
-- Global Context ------------------------------------------------------
------------------------------------------------------------------------
with AT_Stack;
------------------------------------------------------------------------

generic
    ---------------------------------------------------------------------
    -- Implementation Parameters ------------------------------------------
    ---------------------------------------------------------------------

    with package AI_Stack is new AT_Stack (Item => Item);

    type Stack is new AI_Stack.Stack with private;

    ---------------------------------------------------------------------

package AT_Queue.With_Reverse.CT_1 is

    ---------------------------------------------------------------------
    -- Interface ----------------------------------------------------------
    ---------------------------------------------------------------------

    type Queue is new AT_Queue.With_Reverse.Queue with private;

    ---------------------------------------------------------------------
    -- Added Operations ---------------------------------------------------
    ---------------------------------------------------------------------

    procedure Reverse_Queue (
            q : in out Queue
        );

    ---------------------------------------------------------------------
private
    ---------------------------------------------------------------------
    -- Representation Extension -------------------------------------------
    ---------------------------------------------------------------------

    type Queue is new AT_Queue.With_Reverse.Queue with null record;

   ----------------------------------------------------------------------
end AT_Queue.With_Reverse.CT_1;
```

Figure 5.13: Package Specification for AT_Queue.With_Reverse.CT_1

```
------------------------------------------------------------------------
--
-- Component: AT_Queue.With_Reverse.CT_1
-- Relations: implements AT_Queue.With_Reverse, needs AT_Stack
--  Comments: Reverse_Queue reverses a queue using a stack
--
------------------------------------------------------------------------

package body AT_Queue.With_Reverse.CT_1 is

    ------------------------------------------------------------------------
    -- Added Operations ----------------------------------------------------
    ------------------------------------------------------------------------

    procedure Reverse_Queue (
          q : in out Queue
        ) is
          s      : Stack;
          x      : Item;
          length : Integer := 0;
    begin
        Get_Length (q, length);
        while length > 0 loop
            --! alters    q, s, length
            --! consumes  x
            --! maintains reverse(s) * q = reverse(#s) * #q  and
            --!           length = |q|
            --! decreases |q|
            Dequeue (q, x);
            Push (s, x);
            length := length - 1;
        end loop;
        Get_Length (s, length);
        while length > 0 loop
            --! alters    q, s, length
            --! consumes  x
            --! maintains q * s = #q * #s  and  length = |s|
            --! decreases |q|
            Pop (s, x);
            Enqueue (q, x);
            length := length - 1;
        end loop;
    end Reverse_Queue;

    ------------------------------------------------------------------------

end AT_Queue.With_Reverse.CT_1;
```

Figure 5.14: Package Body for AT_Queue.With_Reverse.CT_1

5.7 Other RESOLVE/Ada95 Relationships

The **uses**, **implements**, **needs**, and **extends** relationships each address a fundamental issue of software engineering. These relationships are likely to appear in one form or another in any discipline for component-based software engineering. The **specializes** and **checks** relationships described in this section arise from following the principles of the RESOLVE discipline. The reader unfamiliar with RESOLVE may find these two relationships, especially checks, new and interesting.

5.7.1 The RESOLVE/Ada95 specializes Relationship

As we discussed in Section 5.3, one of the principles of the RESOLVE discipline is that implementations should be fully parameterized. In terms of component relationships, this means that design dependencies should be expressed in terms of **needs**, instead of **uses**, whenever possible. In addition to reducing component coupling, this approach also supports parametric adjustment of the performance characteristics of concrete components [Sit92]. For example, the performance of the queue implementation `AT_Queue.CT_2` shown in Figure 5.7 depends on which `One_Way_List` implementation a client provides as an implementation parameter. The client of this component may "tune" the performance of a concrete component by choosing among different implementations of the components it uses.

The disadvantage of fully parameterized concrete components is that they are more difficult for clients to use. For example, most client programmers who want to use a queue component in their application will not want to be bothered with selecting a list implementation. (The list implementation might also require its own implementation parameters, and so on.) The RESOLVE framework solves this problem with a *specialization component.* A specialization component is a concrete component for which some and usually all of the **needs** relationships have been fixed to **uses** relationships. That is, specialization components are *not* fully-parameterized. A specialization component is produced by internally instantiating implementation parameters of a fully parameterized component and then re-exporting the interface and behavior of the resulting instantiation. A RESOLVE component library may contain several specialization components associated with each fully parameterized kernel concrete component.

The **specializes** relationship is a special case of the **uses** relationship. We define the **specializes** relationship informally as follows:

> Concrete component C_2 **specializes** concrete template C_1 if and only if C_2 **uses** C_1 and all behavior implemented by C_2 is provided by an instantiation of C_1.

A specialization component is a concrete component that **specializes** another concrete component. The component coupling diagram in Figure 5.15 depicts the

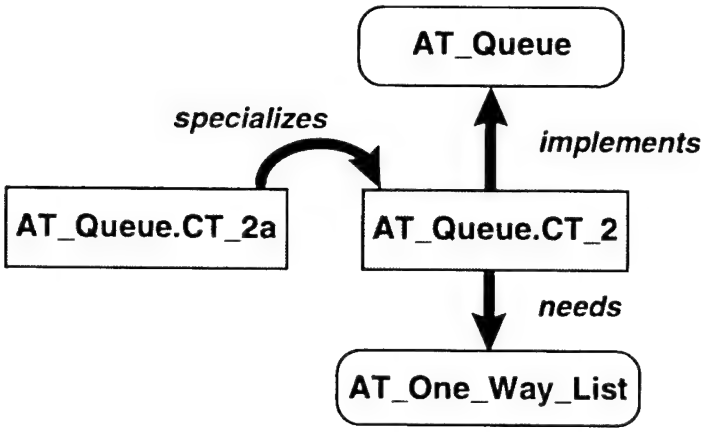


Figure 5.15: The **specializes** Relationship

specializes relationship between two concrete components: `AT_Queue.CT_2a` and `AT_Queue.CT_2`. `AT_Queue.CT_2a` is a specialization component which depends on the fully-parameterized implementation `AT_Queue.CT_2`. In this example, the **specializes** relationship indicates that `AT_Queue.CT_2a` has been created by internally instantiating `AT_Queue.CT_2`. Within `AT_Queue.CT_2a`, one particular component that **implements** `AT_One_Way_List` must be used as the implementation parameter to instantiate `AT_Queue.CT_2`. The interface and behavior exported by the internal instantiation of `AT_Queue.CT_2` is re-exported as the complete interface and behavior of `AT_Queue.CT_2a`.

Several implicit relationships not shown in Figure 5.15 may be deduced from those shown. First, `AT_Queue.CT_2a` **uses** `AT_Queue.CT_2` by the definition of **specializes**. Second, `AT_Queue.CT_2` also **uses** some component which **implements** `AT_One_Way_List` because of the **uses** relationship shown. Third, and most important, `AT_Queue.CT_2a` **implements** `AT_Queue` because of the **implements** relationship shown. Thus, an instance of `AT_Queue.CT_2a` is behaviorally substitutable for an instance of `AT_Queue.CT_2` with respect to `AT_Queue`. The difference between these two implementations of `AT_Queue` is that the non-functional characteristics of `AT_Queue.CT_2a` have been fixed while some of the non-functional characteristics of `AT_Queue.CT_2` may still be adjusted.

A specialization component is encoded in RA95 as a generic child unit package specification. No corresponding package body is needed. The parent unit is the abstract component which the component being specialized (and thus also the specialization component itself) implements. By convention, the name of a specialization component is the same as that of the component it **specializes** appended with a

single letter used to distinguish between multiple specializations of the same component. For example, the name AT_Queue.CT_2c identifies the "third" specialization of the "second" implementation of the queue abstract template.

A specialization component contains a global context section, an interface section, a local instantiations section, and a representation section. Typically all of the parameters of the component being specialized are fixed. In this case, there will be no implementation parameters section and no generic formal parameters. The fixed dependencies on the component being specialized and on the components selected to serve as implementation parameters appear as `with` clauses in the global context section. The interface section of a specialization component declares the exported concrete type as a private type extension of the abstract type exported by the parent unit. This partial view of the exported type assures that a client of this component has visibility to all operations described in the abstract parent unit. The implementations of these operations are provided in the private part and thus no operations need to be declared in the interface section.

The private part of the package contains the local instantiations and representation sections. The local instantiations section contains all package instantiations necessary to supply actual parameters to the final instantiation in this section. The final instantiation creates an instance of the component being specialized. This package instance supplies the concrete type to be re-exported by the specialization component. The representation section consists of a null record extension of the concrete type to be re-exported. This type extension creates the full view of the concrete exported type.

An example of a specialization component is AT_Queue.CT_2a. This component is a concrete template which **specializes** AT_Queue.CT_2 (Figure 5.5) and **uses** the specialization AT_One_Way_List.CT_1a (not shown) to instantiate AT_Queue.CT_2. Note that AT_Queue.CT_2a **implements** AT_Queue. Figure 5.16 shows a component instantiation diagram detailing *how* AT_Queue.CT_2a has been implemented. From the information summarized in this diagram, it would be straightforward for a tool to automatically generate the Ada code for AT_Queue.CT_2a shown in Figure 5.17.

The local instantiations section of AT_Queue.CT_2a includes three package instantiations. The first creates AI_One_Way_List, an instantiation of AT_One_Way_List with the same Item type as AT_Queue. This instantiation is depicted in Figure 5.16 by the line connecting the Item specification parameter of AT_Queue to that of AT_One_Way_List on the left side of Figure 5.16. The second instantiation creates CI_One_Way_List, an instantiation of AI_One_Way_List formed by selecting the "1a" implementation of AT_One_Way_List. This instantiation is depicted by showing the implementation AT_One_Way_List.CT_1a around the specification AT_One_Way_List in the lower left corner of the figure. The final instantiation creates CI_Queue, a concrete instance of AT_Queue.CT_2, the component being specialized. This instantiation is depicted in Figure 5.16 by placing the AT_Queue.CT_2 implementation

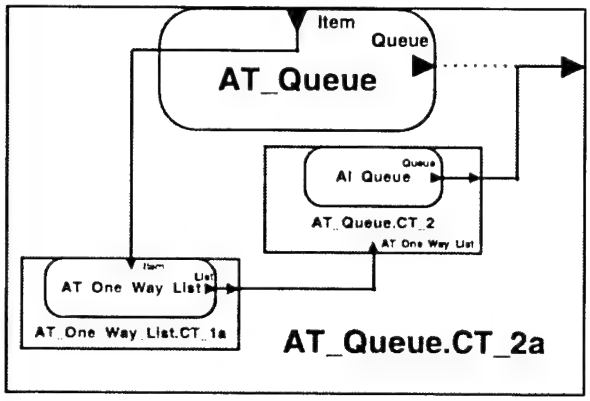


Figure 5.16: A Detailed View of AT_Queue.CT_2a

around the AI_Queue specification and by the line connecting the type exported by AT_One_Way_List.CT_1a to the implementation parameter of AT_Queue.CT_2.

Note that the component name AI_Queue in Figure 5.16 refers to an instance of AT_Queue with the type Item fixed. Within the child unit AT_Queue.CT_2a, the package name AT_Queue implicitly refers to this package instance, an instance of its generic parent unit. In Figure 5.16, the line from the type exported by AT_Queue.CT_2 to the type exported by AT_Queue.CT_2a depicts the inheritance link between these two types. The null record type extension at the end of AT_Queue.CT_2a encodes this inheritance link.

5.7.2 The RESOLVE/Ada95 checks Relationship

This section describes *checking components* and the associated **checks** relationship. Checking components are very useful components within the RESOLVE framework. Their utility largely results from the layered way in which RESOLVE components are designed and implemented.

One of the principles of the RESOLVE discipline is that clients should be responsible for checking the preconditions of operations. This approach avoids unnecessary inefficiency and simplifies component implementations. Furthermore, it respects the contractual relationship expressed by the **implements** relationship. A component implementer is responsible for providing all behavior described by the abstract component and no more. If a concrete component must handle exceptional events, then the abstract template should describe what behavior is required for exceptional events. If a client uses an operation when its precondition is not satisfied, then the concrete

```
------------------------------------------------------------------------
--
-- Component: AT_Queue.CT_2a
-- Relations: specializes AT_Queue.CT_2, uses AT_One_Way_List.CT_1a
--  Comments: -
--
------------------------------------------------------------------------
-- Global Context ------------------------------------------------------
------------------------------------------------------------------------

with AT_Queue.CT_2;
with AT_One_Way_List.CT_1a;

------------------------------------------------------------------------

generic

package AT_Queue.CT_2a is

    ------------------------------------------------------------------
    -- Interface -----------------------------------------------------
    ------------------------------------------------------------------

    type Queue is new AT_Queue.Queue with private;

    ------------------------------------------------------------------

private

    ------------------------------------------------------------------
    -- Local Instantiations ------------------------------------------
    ------------------------------------------------------------------

    package AI_One_Way_List is new
        AT_One_Way_List (Item => Item);

    package CI_One_Way_List is new
        AI_One_Way_List.CT_1a;

    package CI_Queue is new
        AT_Queue.CT_2 (
                AI_One_Way_List => AI_One_Way_List,
                           List => CI_One_Way_List.List
            );
```

Figure 5.17: Abstract Template AT_Queue.CT_2a

```
    ------------------------------------------------------------------
    -- Representation ------------------------------------------------
    ------------------------------------------------------------------

    type Queue is new CI_Queue.Queue with null record;

    ------------------------------------------------------------------

end AT_Queue.CT_2a;
```

Figure 5.18: Abstract Template AT_Queue.CT_2a (Continued)

component exporting that operation is free to do anything or nothing from that point on.

Particularly during the testing and debugging of new implementations, unreliable code may erroneously call operations with violated preconditions. In RESOLVE, the behavior of the implementations is unspecified once a precondition violation takes place. As a result, locating errors while debugging may be very difficult. Thus, checking the preconditions of operations is useful when there is a lack of confidence in the correctness of implementations within a software system. A checking component addresses this problem by checking operation preconditions before calling unprotected operation implementations.

A checking component is a concrete component with characteristics similar to both an extension component and a concrete component. Like an extension component, it adds new functionality to another component. The additional behavior consists of checking to see if a precondition is satisfied, and if not, reporting the violation and halting execution of the program immediately. Unlike an extension component, however, a checking component provides no additional specified behavior. Like a typical concrete component, a checking component provides the interface and behavior specified by an abstract template. However, a checking component does not directly provide the specified functionality it exports. Instead, a checking component uses the abstract template which it **implements** to provide its exported behavior. That is, a checking component is implemented by layering the checking functionality on top of a concrete component that provides the desired specified behavior. Therefore, a single checking component may be used to check any concrete component which **implements** the abstract component it **checks**.

The **checks** relationship expresses part of the dependency between a concrete component and an abstract component. The **checks** relationship may be defined informally as follows:

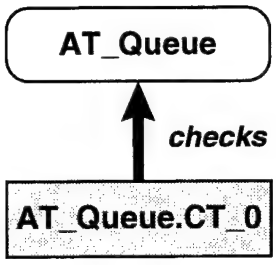


Figure 5.19: The **checks** Relationship

> Concrete component template C **checks** abstract component A if and only if (a) C **implements** A and (b) C immediately reports violations of all operation preconditions described by A.

A checking component is a concrete component which **checks** and **needs** the same abstract component. It is possible to have the relationship C **checks** A without the relationship C **needs** A. However, such a C, one which directly implements the behavior of the component it checks, would not be nearly as useful as a checking component.

The CCD in Figure 5.19 depicts the **checks** relationship between the checking component `AT_Queue.CT_0`, and the abstract component `AT_Queue`. The only precondition specified by `AT_Queue` is that the `Dequeue` operation may not be called with an empty queue object. Therefore, if a client of `AT_Queue.CT_0` tries to dequeue an item from an empty queue, an error report will be issued and the program will halt. Note that in the case of a checking component, the **needs** relationship is omitted from the CCD. A slightly more complex notation which includes template parameters makes explicit this `uses` relationship for checking components.

A checking component is encoded in RA95 as a generic child unit of the abstract template which it checks. By convention, the package name `CT_0` is prefixed by the name of its parent unit. Since the child unit has direct visibility to its parent, the `uses` relationship with its parent is implicit. Thus, the global context section in a checking component is empty. The implementation parameters section contains a single generic formal derived type. The name of this type is the name of the exported concrete type prefixed by the string `Base_`. The actual parameter for this formal type must be a type derived from the abstract type exported by the parent unit. Thus, the type exported by any component which **implements** the parent unit may serve as an actual parameter.

Figure 5.20 shows the RA95-specific component instantiation diagram depicting the parent-child relationship between a kernel abstract component and its concrete

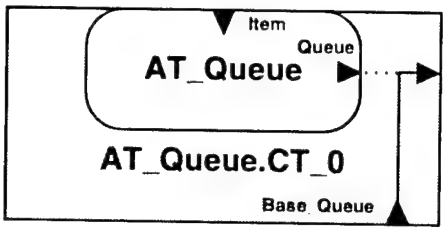


Figure 5.20: Abstract Parent and Concrete Checking Child

checking component. In this example, `AT_Queue.CT_0` is the checking component for `AT_Queue`. The implementation parameter `Base_Queue`, like all other implementation parameters, is shown along the bottom edge of the concrete component. The dotted line conveys the constraint that the actual parameter supplied for `Base_Queue` must be a type derived (possibly indirectly) from the type exported from an instance of `AT_Queue`.

The interface section of a checking component package specification includes the declaration of the concrete exported type. The exported type is a null record extension of the imported generic formal type. Thus, the representation of the component on which the checking component is layered is not altered. The inheritance link established by this type extension is depicted in Figure 5.20 by the line connecting the `Base_Queue` implementation parameter to the type exported by `AT_Queue.CT_0`.

For each operation with a precondition (aside from "true") in the parent unit, the child unit includes a corresponding concrete subprogram signature. These operations override those with matching signatures inherited from the generic type parameter. This is another use of mixin inheritance. While overriding inherited operations is generally a threat to preserving behavioral substitutability, this overriding is safe as long as the RA95 discipline is followed. It is safe because the overriding operation provides the identical specified behavior as the overridden operation, where the precondition holds (it is permitted to have any behavior where the precondition does not hold).

The package body of a checking component provides the implementations of each overriding operation. The body of each overriding operation first checks to see if the precondition for that operation is satisfied. One of RESOLVE's guidelines for the selection of primitive operations is that they include any operations necessary to check all preconditions. Therefore, the code to check the precondition may be layered. Once the check is made, some mechanism must be used to report the error and halt the program, if necessary. The ideal mechanism used depends upon the run-time

environment in which the RA95 programs will be run. Many Ada compilers provide an `Assert` pragma which is useful for this purpose.

If there are no precondition violations, the body of the overriding operation calls the operation it has overridden. This requires use of Ada's *view conversion*. Inside the body of the overriding operation, a formal parameter of the exported type is converted to the type of its parent when used as an actual parameter in the call to the overridden operation.

Figure 5.21 shows the RA95 package specification for the checking component `AT_Queue.CT_0`. Figure 5.22 shows its package body. `AT_Queue.CT_0` is a concrete template which checks the kernel abstract component `AT_Queue` shown in Figure 5.1. The implementation shown here depends on the `Assert` pragma as implemented by the GNAT Ada compiler.

5.8 Instantiation of RESOLVE/Ada95 Components

In this section we briefly explain and provide an example of how RA95 components may be instantiated to form concrete instances which may be used directly in applications. Recall that a concrete component is a subsystem implementation for which all parameters have been fixed. Thus, a concrete component exports a concrete type which may be used directly by another component or application program. In Section 5.7.1, we explained the instantiation of several components within the package specification of `CT_Queue.CT_2a`. These local instantiations were depicted in Figure 5.16. Instantiation of components for use by application programs is similar.

The process of building a concrete instance generally proceeds as follows. First, a kernel abstract instance is produced by an instantiation which binds the specification parameters of a kernel abstract template. Then a kernel concrete instance is produced by an instantiation which binds a specific implementation to the kernel abstract instance. This instantiation may involve supplying actual parameters for implementation parameters. Then, if necessary, the functionality of the kernel concrete instance may be augmented through a sequence of abstract and concrete extensions. For each abstract extension, the type exported from the most recently constructed concrete instance serves as the actual parameter for the specification parameter. Each abstract instance created by an abstract extension must then be supplied with an implementation in the same manner as the kernel abstract instance. Checking concrete instances may be introduced after the kernel concrete instance has been created. A checking component is produced by supplying a concrete instance as an implementation parameter to a checking concrete template.

Figure 5.23 shows a CID depicting the composition of components to form the concrete instance `CI_Enhanced_Integer_Queue_1`. This concrete component provides the behavior specified by `AT_Queue` extended with `AT_Queue_With_Reverse` and

```
------------------------------------------------------------------------
--
-- Component: AT_Queue.CT_0
-- Relations: checks AT_Queue
--  Comments: GNAT -a switch must be on when compiling this and clients
--
------------------------------------------------------------------------
-- Global Context ------------------------------------------------------
------------------------------------------------------------------------

------------------------------------------------------------------------

generic

    ------------------------------------------------------------------
    -- Implementation Parameters -----------------------------------
    ------------------------------------------------------------------

    type Base_Queue is new AT_Queue.Queue with private;

    ------------------------------------------------------------------

package AT_Queue.CT_0 is

    ------------------------------------------------------------------
    -- Interface ---------------------------------------------------
    ------------------------------------------------------------------

    type Queue is new Base_Queue with null record;

    ------------------------------------------------------------------
    -- Overriding Operations ---------------------------------------
    ------------------------------------------------------------------

    procedure Dequeue (
            q : in out Queue;
            x : in out Item
        );

    ------------------------------------------------------------------

end AT_Queue.CT_0;
```

Figure 5.21: Package Specification for Abstract Template AT_Queue.CT_0

```
--------------------------------------------------------------------------
--
-- Component: AT_Queue.CT_0
-- Relations: checks AT_Queue
--  Comments: GNAT -a switch must be on when compiling this and clients
--
--------------------------------------------------------------------------
-- Global Context ------------------------------------------------------
--------------------------------------------------------------------------

--------------------------------------------------------------------------

package body AT_Queue.CT_0 is

    --------------------------------------------------------------------
    -- Interface Operations ----------------------------------------------
    --------------------------------------------------------------------

    procedure Dequeue (
            q : in out Queue;
            x : in out Item
        ) is
            length : Integer := 0;
    begin
        Get_Length (q, length);
        pragma Assert (length > 0,
            "Dequeue pre-condition (q /= empty_string) violated");
        Dequeue (Base_Queue(q), x);
    end Dequeue;

    --------------------------------------------------------------------

end AT_Queue.CT_0;
```

Figure 5.22: Package Body for Abstract Template `AT_Queue.CT_0`

`AT_Queue_With_Replica`[26] (not shown). Note that a component library suitable for production use is likely to contain a wide variety of ready-to-use concrete instances, especially for common data structures such as queues. Therefore, it is unlikely that a component library user would ever need to construct this concrete instance.

One new notation introduced in this diagram is the striped rectangle containing "Integer". This notation is used to distinguish Ada built-in types such as `Boolean`, `Integer`, `Character`, and `Float` from types exported from library components.

[26]The `Replica` operation makes a value (deep) copy of an object. In RESOLVE, this is a usually a layered secondary operation. However, the `Replica` operation required for the built-in type `Integer` is provided by the special package `CI_Scalar_Operations`.

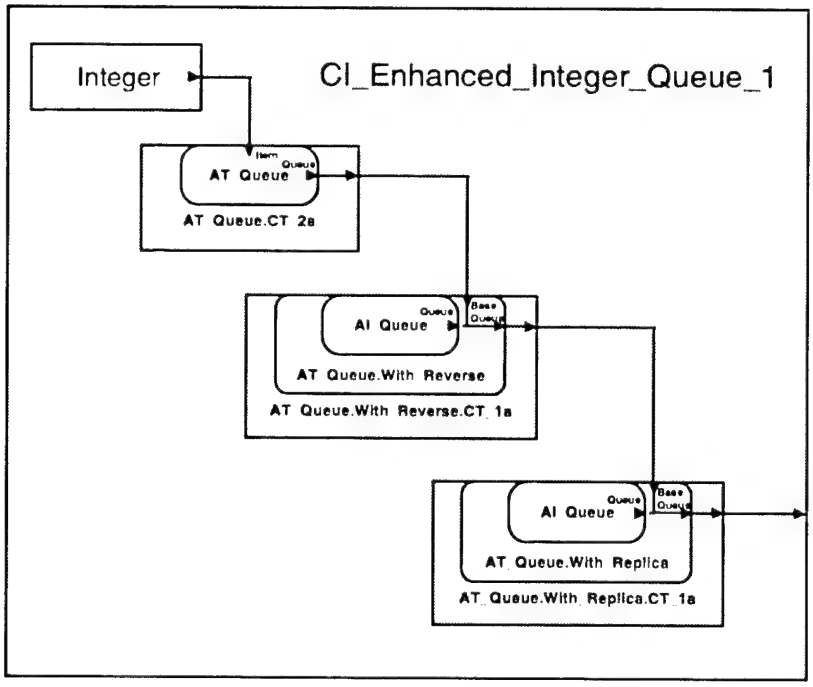


Figure 5.23: A Detailed View of `CI_Enhanced_Integer_Queue_1`

The rightmost arrow touching the exterior boundary of Figure 5.23 represents the `Queue` type exported from `CI_Enhanced_Integer_Queue_1`. The specified behavior of objects of the exported `Queue` type is the union of the specifications named by the clear rounded boxes through which the chain of arrows travels. The implementations providing the actual behavior of objects of this type are named by the shaded rectangular boxes (including that of the built-in type `Integer`) through which the chain of arrows travels.

The Ada package specification which encodes `CI_Enhanced_Integer_Queue_1` is shown in Figure 5.24. Figure 5.23 graphically depicts this code. The concrete instance `CI_Scalar_Operations` provides `Swap` and `Replica` procedures for Ada built-in types such as `Integer`.

5.9 RESOLVE/Ada95 Design Issues

The development of the RA95 approach presented in this chapter involved exploring many challenging design issues and making some compromises. Type extension and hierarchical library units, in particular, presented opportunities and challenges not addressed in the development of RA83. While Ada's good support for modularity

```
------------------------------------------------------------------------
--
-- Component: CI_Enhanced_Integer_Queue_1
-- Relations: uses AT_Queue.CT_2a, uses AT_Queue.With_Reverse.CT_1a,
--            uses AT_Queue.With_Replica.CT_1a
--  Comments: CI_Scalar_Oparations provides Swap for built-in Integer
--
------------------------------------------------------------------------
-- Global Context ------------------------------------------------------
------------------------------------------------------------------------

with CI_Scalar_Operations;
use  CI_Scalar_Operations;

with AT_Queue.CT_2a;
with AT_Queue.With_Reverse.CT_1a;
with AT_Queue.With_Replica.CT_1a;

------------------------------------------------------------------------

package CI_Enhanced_Integer_Queue_1 is

    ---------------------------------------------------------------------
    -- Local Instantiations -----------------------------------------------
    ---------------------------------------------------------------------

    -- instantiate AT_Queue with Integer

    package AI_Integer_Queue is new
        AT_Queue(Item => Integer);

    ---------------------------------------------------------------------

    -- implement AI_Integer_Queue with CT_2a

    package CI_Integer_Queue is new
        AI_Integer_Queue.CT_2a;

    ---------------------------------------------------------------------

    -- extend AI_Integer_Queue with Reverse

    package AI_Integer_Queue_With_Reverse is new
        AI_Integer_Queue.With_Reverse (
            Base_Queue => CI_Integer_Queue.Queue);
```

Figure 5.24: Package Specification for `CI_Enhanced_Integer_Queue_1`

```
   ------------------------------------------------------------------

   -- implement AI_Queue_With_Reverse with CT_1a

   package CI_Integer_Queue_With_Reverse is new
       AI_Integer_Queue_With_Reverse.CT_1a;

   ------------------------------------------------------------------

   -- extend AI_Integer_Queue with Replica

   package AI_Integer_Queue_With_Replica is new
       AI_Integer_Queue.With_Replica (
           Base_Queue => CI_Integer_Queue_With_Reverse.Queue);

   ------------------------------------------------------------------

   -- implement AI_Queue_With_Replica with CT_1a

   package CI_Integer_Queue_With_Reverse_And_Replica is new
       AI_Integer_Queue_With_Replica.CT_1a;

   ------------------------------------------------------------------
   -- Interface -----------------------------------------------------
   ------------------------------------------------------------------

   type Queue is new CI_Integer_Queue_With_Reverse_And_Replica.Queue
       with null record;

   ------------------------------------------------------------------

end CI_Enhanced_Integer_Queue_1;
```

Figure 5.25: Package Specification for `CI_Enhanced_Integer_Queue_1` (Continued)

and genericity makes it possible to apply much of the RESOLVE discipline using Ada. RESOLVE and Ada are far from being a perfect match. In this section, we discuss some of the major issues faced in the development of RA95.

5.9.1 Initialization of Built-in Scalars

Dealing with Ada's built-in scalar types represented a particularly challenging problem. In Ada, controlled types may be used to provide automatic initialization and finalization for all *user-defined* types. All types derived from `Ada.Finalization`'s `Controlled` or `Limited_Controlled` types have automatic initialization and finalization. Except for access types which must be initialized to null, Ada's built-in scalars

do not have any automatic initialization. In fact, built-in scalars may have initial values that are *invalid representations* of their type [Int95b, §3.3.1(21)].

If Ada required a scalar to be initialized to a *valid* value of its type (not necessarily a particular fixed value), then for reasoning purposes, the initial value of a scalar could be assumed to be a specific *undetermined* default value and uninitialized scalars could have been used in RA95. In this case, scalar types would only lack a potentially useful initial value. However, since uninitialized scalars may have invalid representations, they cannot be passed as arguments to operations since this might raise a `Constraint_Error` or `Program_Error` at runtime [Int95b, §13.9.1(9)].

In RESOLVE, every variable is initialized to a value of its type at the beginning of its scope (upon creation). A common idiom in RESOLVE is to swap the values of a local variable and a consumes mode parameter at the beginning of an operation. This swap is done to obtain an initial-valued object to return for the consumes mode parameter and to ensure finalization of the consumed object before the return. Using uninitialized scalars, such a call to `Swap` could cause a run-time error.

Ada's `Normalize_Scalars` pragma defined in the Safety and Security Annex of [Int95b] requires variables of each scalar type to be initialized to a specific documented value. However, the *implementation advice* for this pragma recommends that the initial value be an *invalid value* for that type, if possible [Int95b, §H.1(1)]. Thus, `Normalize_Scalars`, which is intended to make it easier to detect use of scalars before programmer initialization, advises that compilers do exactly the opposite of what RA95 needs upon initialization. Therefore, `Normalize_Scalars`, when implemented faithfully, is of no use for implementing RA95.

One approach to built-in types, adopted by RESOLVE, is to eliminate built-in types from the language [Har90, §3.3.2]. Unfortunately, this solution cannot be used with Ada or C++ which rely heavily on built-in types, and is impractical when the syntactic sugar that comes with built-in types cannot be duplicated for user-defined replacements.

The solution we chose was to alter compiler source code so that scalars are automatically initialized. We modified the implementation of the `Normalize_Scalars` pragma in the publicly available GNAT source code. The modified compiler satisfies the requirements of the language definition, although it is in direct opposition to the implementation advice provided. This is not an ideal approach due to portability issues, but it does not result in the awkward coding style and inefficiencies of the alternatives.

5.9.2 Limitations of Child Units

The use of generic child units for encoding the **implements** and **extends** relationships has several advantages. As discussed in Section 5.4, concrete component has direct visibility of any specification parameters and other elements within its generic

parent unit. Also, the child unit naming convention is convenient. Using child units, however, does present some limitations. Ada does not allow the instantiation of a parent unit within a child unit of that parent. The parent unit is within the scope of the child unit, thus making such an instantiation recursive, and Ada does not support recursive instantiation of generic units. As a result, a concrete component cannot create and use an instantiation of a sibling unit for which the parent unit has different specification parameters. This precludes specialization components that fix one or more specification parameters from being child units of the kernel concept that they specialize. While partial instantiation of only implementation parameters works in some cases, there are subtle situations where mutual recursion of instantiated units can prevent an instantiation that is legal in RESOLVE.

As an example of this problem, consider the local instantiation of `CI_One_Way_List` shown in Figure 5.17 on page 141. If the implementation, `AI_One_Way_List.CT_1a` in this case, were built using an implementation of `AT_Queue`, then the instantiation of `CI_One_Way_List` would include an instantiation of `AT_Queue`. Therefore, any instantiation of `AT_Queue.CT_2` would include an instantiation of **AT_Queue** which is not allowed in Ada. In general, the recursion among two or more components could be hidden in deeply layered implementations. While cases of mutual recursion such as this might not arise frequently, general solutions for avoiding it tend to be too overly constraining.

5.10 Chapter Summary

In this chapter we demonstrated how the behavioral relationships defined in Chapter 3 can be encoded in Ada. In doing so, we also presented most aspects of the RESOLVE/Ada95 discipline for software component engineering. Ada is better equipped to encode these relationships than most programming languages due to its strong support for modularity and parametric polymorphism. However, new language mechanisms, such as type extension (inheritance) and hierarchical libraries (component extension), added to Ada in 1995, have also proven useful for encoding component relationships.

As with languages is the Module-2 family, but unlike most other languages, Ada components (packages) must explicitly name any components upon which they depend. Ada's `with` context clause thus serves well for encoding the **uses** relationship. We use Ada's abstract types and abstract operations to encode the structural aspects of an abstract component. Behavioral specifications are recorded in structured comments using the RESOLVE specification notation. Ada's type extension (single inheritance) and child unit mechanisms are used in conjunction to encode both the **implements** and **extends** relationships. We use mixin inheritance to encode multiple dependencies with extension implementations. The **needs** relationship is encoded

in Ada using a pair of related generic formal parameters and a `with` clause encoding the **uses** relationship between the concrete template that **needs** the abstract component.

In Section 5.7, we discussed the **specializes** and **checks** relationships which are somewhat unique to the RESOLVE approach. These are both special cases of the **implements** relationship. We concluded this chapter with a discussion of several RA95 design issues. Even with its extensive assortment of language mechanisms, Ada does not provide an ideal level of support for the RESOLVE approach to component-based software engineering.

CHAPTER 6

CONCLUSION

In this chapter we summarize the research conducted for this dissertation and present conclusions drawn from it. We then present the contributions of this research to the field of computer science, and conclude with a discussion of areas for future work.

6.1 Summary and Conclusions

This dissertation defends the thesis that component-level maintenance of software systems may be based on a small set of behavioral and dependency relationships between software components, and that these relationships can be encoded with the language mechanisms provided by modern programming languages, although not as easily as should be possible. Chapters 2 and 3 address the first part of this thesis. Chapters 4 and 5 address the second part.

In Chapter 2, we developed a relatively simple set theoretic model of behavioral relationships between software components. The model does not depend on a specific language syntax or semantics. Instead, it assumes that a language syntax and semantics are defined. Then the model defines behavioral relationships between components, which may be specifications or implementations, either of which may be parameterized or not. The relations defined, **imps**, **exts**, **uses**, and **needs**, are used to model behavioral conformance and dependencies between components.

Chapter 3 explains how the relations defined in Chapter 2 may be used for component-level maintenance of software systems. In order to remove one component from a system and replace it with a behaviorally compatible component, components must be designed with two relationships clearly documented. First, each component in the system that **uses** the component to be replaced must state its behavioral requirement for a suitable implementation. This is the role played by the **needs** relationship. Second, each component should state its behavioral conformance to one or more specifications. This is the role played by the **implements** relationship. The **extends** relationship is important for adding new functionality to components, while

maintaining conformance to existing specifications, and thus minimizing the effects of changes.

Modern programming languages do not provide ideal support for encoding the behavioral relationships we define. Chapter 4 describes a variety of approaches to using the mechanisms of modern programming languages to encode these relationships. In Chapter 5, we demonstrate specifically how the relationships defined in Chapter 3 can be encoded in Ada. Chapter 5 also presents the RESOLVE/Ada95 discipline for software component engineering.

6.2 Contributions

The focus of this research has been to define a set of behavioral relationships between software components and to investigate ways in which these relationship may be encoded using modern programming languages. The primary contributions of this research to the field of computer science are as follows:

A Model of Software Component Relationships

The model of software component relationships developed during this research and defined in Sections 2.2-2.4 provides a single formal framework capturing the semantics of relationships between executable program components, specifications, and templates. The chief leverage gained by using the model is that the *meaning* specifications and templates may be understood in terms of the semantics of operational program components rather than just as syntactic transformations of strings of characters. The model is independent of the language used to encode components and the formalisms used to verify the correctness of an implementation.

Definition of Relationships Supporting Component-Level Maintenance

The component relationships defined in Chapter 3 serve as a basis for component-level maintenance of software because they allow dependencies between components to be stated in terms of *behavioral* requirements, rather than purely syntactic requirements. This allows implementation components to be decoupled from each other prior to system integration and promotes a clear distinction between design dependencies and integration dependencies. The examples presented in Chapters 3 and 5 demonstrate how these relationships may be used in practice to support the well-established software engineering principles of modularity, information hiding, polymorphism, and extendibility.

The RESOLVE/Ada95 Discipline

The RA95 discipline for component-based software development, presented in Chapter 5, was developed as part of the research effort documented in this dissertation. RA95 provides a way for software engineers to apply the principles of RESOLVE in a well-supported and widely available programming language suitable for development of large complex software systems. In addition to providing a concrete example of how component relationships may be encoded, RA95 illustrates new and innovate uses of Ada's unique language mechanisms. In particular, RA95 demonstrates how parametric polymorphism (in the form of Ada generics) and subtype polymorphism (in the form of Ada type extension) may be used in combination to develop well-encapsulated extendible template components. Also, the component instantiation diagrams presented in Chapter 5 should serve as a useful aid for explaining and generating often complex compositions of Ada components.

6.3 Future Research

Future work in the area of software component relationships might progress in several directions. The following sections each discuss a potential area for further research.

Applying the Model to Physical Components

An interesting aspect of the component relationship model presented in Chapter 2 is that appears general enough to apply to physical components as well as software components. While physical systems are not symbolic, they do exhibit behaviors and their design documents are symbolic. Figure 2.1 on page 15 suggests how the **implements** and **needs** relationships might be applied in physical systems. Using a single behavioral framework to describe both software and hardware artifacts might prove useful in dealing with formal models of embedded systems, where the physical system being controlled and the embedded software must be analyzed and designed together. Such a framework might also lead to a better understand the similarities between well-engineered physical systems and well-engineered software systems.

Extending The Model

One of the strengths of the component relationship model presented in Chapter 2 is its relative simplicity. The model is expressive enough to capture the second-order nature of templates which matches the full capability of templates in programming languages such as Ada and C++. Sitaraman has pointed out, however, a practical need for allowing (uninstantiated) template components as parameters to template components [Sit92]. Such "higher order" component compositions are expressible

within the ACTI model, but not within the model presented in Chapter 2. Extending the component relationship model to allow expression of higher order compositions would be a worthy avenue of further research.

Component Relationship-Based Tools

The component relationships presented in this dissertation should be very useful for organizing and using software component libraries. Current navigation tools such as "class browsers", are based on direct coupling relationships, including inheritance links. A library navigation tool based on behavioral (semantically significant) relationships should be more useful, especially for integration of existing components and component-level system maintenance. Designing and implementing a tool based on the relationships presented in Chapter 3 would likely be a useful direction for further work.

Another interesting effort would be the design and development of a component composition tool that generates instantiation code, such at that shown in Figures 5.24 and 5.25 on pages 149 and 150, through graphical manipulation of a corresponding component instantiation diagram, such as that shown in Figure 5.23 on page 148.

Further Developing RESOLVE/Ada95

There are a number of avenues for further development of RA95. Initial efforts to include run-time selectable (dynamically bound) components in RA95 (including efforts by Falis [Fal95]) were not fully successful due the complexity of code required and problems with early Ada95 compilers. With improved Ada95 compilers and experience with parallel efforts in RESOLVE/C++, it may be worth re-investigating this area of research.

Despite Ada's mismatches with the RESOLVE language, ensuring that RA95 code is legal (portable) Ada code thus far has been a priority in the development of RA95. Nevertheless, the public availability of well-documented source code for the GNAT compiler offers the opportunity to modify the RA95 source language so that it is more suitable for RESOLVE-style components. As discussed in Section 5.9.1, the GNAT compiler was modified to automatically initialize scalars. Modifying GNAT by adding `Swap` (perhaps as infix ":=:") as an intrinsic operation for built-in scalars, adding it to **Ada.Finalization**, and allowing "in out" parameters for functions, would make RA95 source code much simpler and the generated object code more efficient.

BIBLIOGRAPHY

[AGBH77] A. L. Amber, D. I. Good, W. F. Burger, and C. G. Hoch. GYPSY: A language for specification and implementation of verifiable programs. *ACM SIGPLAN Notices*, 12(3):1–10, March 1977.

[BLM96] Joseph A. Bank, Barbara Liskov, and Andrew C. Meyers. Parameterized types and java. Technical Report MIT LCS TM-553, Laboratory For Computer Science, Massachusetts Institute of Technology, Cambridge, MA, 1996.

[Boo87] Grady Booch. *Software Components with Ada.* Benjamin/Cummings, Menlo Park, CA, 1987.

[Boo90] Grady Booch. Design of the C++ Booch components. In *ECOOP/OOPSLA'90 Conference Proceedings*, pages 1–11, New York, NY, 1990. ACM.

[Boo94] Grady Booch. *Object-Oriented Analysis and Design With Applications.* Benjamin/Cummings, Menlo Park, CA, 2nd edition, 1994.

[BR97] Gerald Baumgartner and Vincent F. Russo. Implementing signatures for C++. *ACM Transactions on Programming Languages and Systems*, 19(1):153–187, January 1997.

[BS92] B. Banner and E. Schonberg. Assessing Ada9X OOP: Building a reusable components library. In Charles B. Engle, editor, *TRI-Ada'92 Conference Proceedings*, pages 79–90, Orlando, Florida, 1992.

[Bud91] Timothy Budd. *An Introduction To Object-Oriented Programming.* Addison-Wesley, Reading, MA, 1991.

[CE95] Martin D. Carroll and Margaret A. Ellis. *Designing And Coding Reusable C++.* Addison-Wesley, Reading, MA, 1995.

[Cla95] Robert G. Clark. Type safety and behavioral inheritance. *Information and Software Technology*, 37(10):539–545, 1995.

[Coo90] William R. Cook. Inheritance is not subtyping. In *Proceedings of the ACM Conference on Principles of Programming Languages (POPL'90)*, pages 125–135. ACM Press, 1990.

[Cox86] Brad J. Cox. *Object-Oriented Programming*. Addison-Wesley, Reading, MA, 1986.

[CW85] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys*, 17(4):471–522, 1985.

[Dep83] Department of Defense, Ada Joint Program Office. *Reference Manual for the Ada Programming Language, ANSI/MIL-STD-1815A*, 1983.

[DL96] Krishna Kishore Dhara and Gary T. Leavens. Forcing behavioral subtyping through specification inheritance. In *Proceedings of the 18th International Conference on Software Engineering*, pages 258–267. IEEE Computer Society Press, March 1996.

[Edw90] Stephen H. Edwards. An approach for constructing reusable software components in Ada. IDA Paper P-2378, Institute for Defense Analyses, Alexandria, VA, September 1990.

[Edw93] Stephen H. Edwards. Inheritance: One mechanism, many conflicting uses. In Larry Latour, editor, *Proceedings of the Sixth Annual Workshop on Software Reuse*, November 1993.

[Edw95] Stephen Hilary Edwards. *A Formal Model of Software Subsystems*. PhD thesis, The Department of Computer and Information Science, The Ohio State University, Columbus, Ohio, 1995.

[Edw96] Stephen H. Edwards. Representation inheritance: A safe form of "white box" code inheritance. In *The Fourth International Conference on Software Reuse*. IEEE Computer Society Press, April 1996.

[EHM091] George W. Ernst, Raymond J. Hookway, James A. Menegay, and William F. Ogden. Modular verification of Ada generics. *Computer Languages*, 16(3/4):259–280, 1991.

[EHO94] George W. Ernst, Raymond J. Hookway, and William F. Ogden. Modular verification of data abstractions with shared realizations. *IEEE Transactions on Software Engineering*, 20(4):288–307, 1994.

[End77] Herbert B. Enderton. *Elements of Set Theory*. Academic Press, Inc., San Diego, California, 1977.

[Fal95] Ed Falis. RESOLVE/Ada 95 Mappings (Draft). Unpublished draft, May 1995.

[GH93] John V. Guttag and James J. Horning. *Larch Languages and Tools for Formal Specification.* Springer-Verlag, 1993.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns.* Addison-Wesley, Reading, Massachusetts, 1995.

[GHW85] John V. Guttag, James J. Horning, and Jeannette M. Wing. Larch family of specification languages. *IEEE Software*, 2(5):24–36, September 1985.

[Gog84] Joseph A. Goguen. Parameterized programming. *IEEE Transactions on Software Engineering*, 10(5):528–543, September 1984.

[Gog86] Joseph A. Goguen. Reusing and interconnecting software components. *IEEE Computer*, 18(2):16–28, February 1986.

[Har82] Samuel P. Harbison. *Modula-3.* Prentice Hall, Englewood Cliffs, New Jersey, 1982.

[Har90] Douglas E. Harms. *The Influence of Software Reuse on Programming Language Design.* PhD thesis, Dept. of Computer and Information Science, The Ohio State University, Columbus, OH, 1990.

[Hey95] Wayne D. Heym. *Computer Program Verification: Improvements For Human Reasoning.* PhD thesis, Dept. of Computer and Information Science, The Ohio State University, Columbus, OH, 1995.

[HLOW94] Wayne D. Heym, Timothy J. Long, William F. Ogden, and Bruce W. Weide. Mathematical foundations and notation of RESOLVE. Technical Report OSU-CISRC-8/94-TR45, The Ohio State University, Columbus, OH, August 1994.

[Hoa72] C. A. R. Hoare. Proof of correctness of data representations. *Acta Informatica*, 1:271–781, 1972.

[Hol92] Joseph E. Hollingsworth. *Software Component Design-for-Reuse: A Language-Independent Discipline Applied to Ada.* PhD thesis, The Department of Computer and Information Science, The Ohio State University, Columbus, Ohio, 1992.

[Hol97] Joseph E. Hollingsworth. Rethinking our answers to fundamental engineering dilemmas. In Larry Latour, editor, *Proceedings of the Eighth Annual Workshop on Software Reuse*, 1997.

[HW91] Douglas E. Harms and Bruce W. Weide. Copying and swapping: Influences on the design of reusable software components. *IEEE Transactions on Software Engineering*, 17(5):424–435, May 1991.

[Int95a] Intermetrics, Inc., Concord, Massachusetts. *Ada 95 Rationale*, January 1995.

[Int95b] Intermetrics, Inc. *Ada 95 Reference Manual. ANSI/ISO/IEC-8652:1995*, January 1995.

[Jon90] Cliff B. Jones. *Systematic Software Development Using VDM*. Prentice Hall International (UK) Ltd, Hertfordshire, England, 2nd edition, 1990.

[Kem95] Magnus Kempe. The composition of abstractions: Evolution of software component design with Ada 95. In *TRI-Ada'95 Conference Proceedings*, pages 391–405, New York, NY, 1995. ACM.

[Kro88] Joan Krone. *The Role of Verification in Software Reusability*. PhD thesis, The Department of Computer and Information Science, The Ohio State University, Columbus, Ohio, 1988.

[LBR96] Konstantin Läufer, Gerald Baumgartner, and Vincent F. Russo. Safe structural conformance for java. Technical Report CSD-TR-96-077, Purdue University, West Lafayette, IN, 1996.

[LBSB80] B. P. Lientz, P. Bennet, E. B. Swanson, and E. Burton. *Software Maitenance Management*. Addison-Wesley, Reading, 1980.

[LCD+91] Barbara Liskov, Dorthy Curtis, Mark Day, Sanjay Ghemawhat, Robert Gruber, Paul Johnson, and Andrew C. Meyers. *Theta Reference Manual (Preliminary Version)*. MIT Laboratory for Computer Science, Cambridge, Massachusetts, February 1994.

[LDGM95] Barbara Liskov, Mark Day, Robert Gruper, and Andrew C. Meyers. Subtypes vs. where clauses: Constraining parametric polymorphism. In *OOPSLA '95*, pages 156–168, Austin, TX, 1995.

[LG86] Barbara Liskov and John Guttag. *Abstraction and Specification in Program Development*. The MIT Electrical Engineering and Computer Science Series, MIT Press, Cambridge, MA, 1986.

[LvHKB087] D. Luckham, F. W. von Henke, B. Krieg-Brückner, and O. Owe. *ANNA: A Language for Annotating Ada Programs*. Springer-Verlag, New York, NY, 1987.

[LW90] Gary Leavens and W. Weihl. Reasoning about object-oriented programs that use subtypes. In *ECOOP/OOPSLA'90 Conference Proceedings*, pages 212–223, New York, NY, 1990. ACM.

[LW94] Barbara H. Liskov and Jeannette M. Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems*, 16:1811–1841, November 1994.

[McI76] M. D. McIlroy. Mass-produced software components. In J. M. Buxton, P. Naur, and B. Randell, editors, *Software Engineering Concepts and Techniques*, pages 88–98. Petrocelli/Charter, 1976.

[Mey86] Bertrand Meyer. Genericity versus inheritance. In *OOPSLA'86 Proceedings*, pages 391–405, New York, NY, 1986. ACM Press.

[Mey87] Bertrand Meyer. Reusability: The case for object-oriented design. *IEEE Software*, pages 201–215, March 1987.

[Mey88] Bertrand Meyer. *Object-Oriented Software Construction.* Prentice Hall, New York, NY, 1988.

[Mey94] Bertrand Meyer. *Reusable Software: The Base Object-Oriented Component Libraries.* Prentice Hall International, Hertfordshire, UK, 1994.

[Mey96] Bertrand Meyer. The many faces of inheritance: A taxonomy of taxonomy. *IEEE Computer*, 29(5):105–108, May 1996.

[MW90] S. Muralidharan and Bruce W. Weide. Should data abstraction be violated to enhance software reuse? In *Proceedings of the 8th Annual National Conference on Ada Technology*, pages 515–524, Atlanta, GA, March 1990. ANCOST, Inc.

[OW97] Martin Odersky and Philip Wadler. Pizza into java: Translating theory into practice. In *Proceedings POPL '97*, pages 146–159, Paris, January 15-17 1997.

[Par72] David L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, December 1972.

[Pre97] Roger S. Pressman. *Software Engineering: A Practitioner's Approach.* McGraw-Hill, New York, NY, 4th edition, 1997.

[RW92] Martin Reiser and Niklaus Wirth. *Programming In Oberon: Steps Beyond Pascal And Modula.* Addison-Wesley, Reading, MA, 1992.

[Sei94] Ed Seidewitz. Genericity versus inheritance reconsidered: Self-reference using generics. In *OOPSLA'94 Proceedings*, pages 153–163, New York, NY, 1994. ACM Press.

[Sel89] Richard W. Selby. Quantitative studies of software reuse. In Ted J. Biggerstaff and Alan J. Perlis, editors, *Software Reusability, Volume II: Applications and Experience*, pages 213–233. ACM Press, New York, NY, 1989.

[SG95] Raymie Stata and John V. Guttag. Modular reasoning in the presence of subclassing. In *OOPSLA'95 Conference Proceedings*, pages 200–214, New York, NY, 1995. ACM.

[Sha81] Mary Shaw. *ALPHARD: Form and Content*. Springer-Verlag, New York, NY, 1981.

[Sit92] Murali Sitaraman. Performance-parameterized reusable software components. *International Journal of Software Engineering and Knowledge Engineering*, 2(4):567–587, December 1992.

[SMC74] W. P. Stevens, G. J. Meyers, and L. L. Constantine. Structured design. *IBM Systems Journal*, 13(2):115–139, 1974.

[Sny86] Alan Snyder. Encapsulation and inheritance in object-oriented programming languages. In *OOPSLA'86 Conference Proceedings*, pages 38–45, New York, NY, 1986. ACM.

[SOM94] Clemens Szyperski, Stephen Omohundro, and Stephan Murer. Engineering a programming language: The type and class system of sather. In Jürg Gutknecht, editor, *Programming Languages and System Architectures*, volume 782 of *Lecture Notes in Computer Science*, pages 208–227. Springer Verlag, March 1994.

[Str93] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, Reading, MA, 2nd edition, 1993.

[SW94] Murali Sitaraman and Bruce W. Weide, editors. Special feature: Component-based software using RESOLVE. *ACM SIGSOFT Software Engineering Notes*, 19(4):21–67, 1994.

[SWH93] Murali Sitaraman, Lonnie R. Welch, and Douglas E. Harms. On specification of reusable software components. *International Journal of Software Engineering and Knowledge Engineering*, 3(2):207–229, 1993.

[SWO97] Murali Sitaraman, Bruce W. Weide, and William F. Ogden. On the practical need for abstraction relations to verify abstract data type representations. *IEEE Transactions on Software Engineering*, 23(3):157–170, March 1997.

[Tai96] Antero Taivalsaari. On the notion of inheritance. *ACM Computing Surveys*, 28(3):438–479, September 1996.

[Tra95] Will Tracz. *Confessions of a Used Program Salesman: Institutionalizing Software Reuse.* Addison-Wesley, Reading, MA, 1995.

[Ull95] Jeffrey D. Ullman. *Elements of ML Programming.* Prentice Hall, Englewood Cliffs, NJ, 1995.

[WEH+96] Bruce W. Weide, Stephen H. Edwards, Wayne D. Heym, Timothy J. Long, and William F. Ogden. Characterizing observability and controllability of software components. In *The Fourth International Conference on Software Reuse.* IEEE Computer Society Press, April 1996.

[Wei97] Bruce W. Weide. Software component engineering. Unpublished Draft, 1997.

[Wel95] David Weller. The Ada 95 Booch components. In *TRI-Ada'95 Conference Tutorial Proceedings*, pages 175–223, New York, NY, 1995. ACM. (See http://www.ocsystems.com/booch/ for code.).

[WH92] Bruce W. Weide and Joseph E. Hollingsworth. Scalability of reuse technology to large systems requires local certifiability. In Larry Latour, editor, *Proceedings of the Fifth Annual Workshop on Software Reuse*, October 1992.

[WHH94] Bruce W. Weide, Wayne D. Heym, and Joseph E. Hollingsworth. Reverse engineering of legacy code is intractable. Technical Report OSU-CISRC-10/94-TR55, The Ohio State University, Columbus, OH, 1994.

[Win90] Jeannette M. Wing. A specifier's introduction to formal methods. *IEEE Computer*, 23(9):8–24, September 1990.

[Wir82] Niklaus Wirth. *Programming in Modula-2.* Springer-Verlag, New York, NY, 1982.

[WOZ91] Bruce W. Weide, William F. Ogden, and Stuart H. Zweben. Reusable software components. In M. C. Yovits, editor, *Advances in Computers*, volume 33, pages 1–65. Academic Press, 1991.

[ZEWH95] Stuart H. Zweben, Stephen H. Edwards, Bruce W. Weide, and Joseph E. Hollingsworth. The effects of layering and encapsulation on software developement cost and quality. *IEEE Transactions on Software Engineering*, 21(3):200–208, 1995.